Technical Report of Computer Science, Swansea University
CSR-2-2009

AVoCS'09

PreProceedings of the
Ninth International Workshop on
Automated Verification of Critical Systems

Gregynog Conference Centre

Swansea University
Prifysgol Abertawe

Editors: Liam O'Reilly and Markus Roggenbach
AVoCS 2009 Home Page: http://www.cs.swansea.ac.uk/avocs09/

Preface

AVOCS, the workshop on Automated Verification of Critical Systems, is an annual meeting that brings together researchers and practitioners to exchange new results on tools and techniques for the verification of critical systems. Topics of interest cover all aspects of automated verification, including model checking, theorem proving, abstract interpretation, and refinement pertaining to various types of critical systems (safety-critical, security-critical, business-critical, performance-critical, etc.). Contributions that describe different techniques, or industrial case studies are encouraged.

Previous AVOCS workshops were held at the University of Oxford (2001 and 2007), the University of Birmingham (2002), the University of Southampton (2003), the Royal Society in London (2004), the University of Warwick (2005), LORIA, Nancy (2006), and the University of Glasgow (2008).

In 2009 authors from twelve countries, namely Algeria, Egypt, Estonia, France, Germany, India, Italy, Japan, The Netherlands, Portugal, the United Kingdom, and the United States, submitted papers to AVOCS.

AVOCS 2009 received 17 submissions for Full Papers, out of which 12 were selected for presentation at the workshop. Furthermore, AVOCS received 10 submissions for Short Contributions all of which were accepted. The selection process was carried out by the Program Committee, taking into account the originality, quality, and relevance of the material presented in each submission. The selected and revised Full Papers and Short Contributions are included in this volume, together with the contributions from the invited speakers Ulrich Berger and Christoph Lüth. After the workshop the authors of Full Papers as well as the invited speakers will have the opportunity to react to feedback and discussions at the workshop: the final revision of their articles will appear in the Proceedings of AVOCS 2009 to be published in the Electronic Communications of the European Association of Software Science and Technology EASST.

We wish to thank all authors who submitted their papers to AVOCS 2009, the Program Committee for its excellent work, and the referees who supported the Program Committee in the evaluation and selection process.

We are grateful to the Department of Computer Science at Swansea University for hosting the event and thank IT Wales for sponsoring AVOCS 2009. We gratefully acknowledge the excellent cooperation with the Gregynog Conference Centre and the use of EasyChair, the conference management system by Andrei Voronkov.

September 2009

Liam O'Reilly
Markus Roggenbach

Organisation

AVoCS 2009 Program Committee

David Aspinall	University of Edinburgh, Scotland, UK
Muffy Calder	University of Glasgow, Scotland, UK
Michael Goldsmith	University of Warwick, UK
Constance Heitmeyer	Naval Research Laboratory, Washington DC, USA
Gerald Luettgen	University of Bamberg, Germany
Eric Mercer	Brigham Young University, Utah, USA
Stephan Merz	INRIA, Nancy, France
Alice Miller	University of Glasgow, Scotland, UK
Paritosh Pandya	IIT Mumbai, India
Silvio Ranise	University of Verona, Italy
Markus Roggenbach (Chair)	Swansea University, Wales, UK
Bill Roscoe	University of Oxford, UK
Mark Ryan	University of Birmingham, UK
Helen Treharne	University of Surrey, UK
Markus Wenzel	TU München, Germany
Sergio Yovine	Verimag, France

AVoCS 2009 Reviewers

Arnold Beckmann	Swansea University, Wales, UK
Ulrich Berger	Swansea University, Wales, UK
Jan Calta	Humboldt University Berlin, Germany
Jean-Christophe Filliatre	Université Paris Sud, France
Dominique Mery	INRIA, Nancy, France
Jan Tobias Muehlberg	University of Bamberg, Germany
Dirk Pattinson	Imperial College London, UK
Heiko Schmidt	University of Bamberg, Germany
Stefano Tonetta	FBK-Irst, Italy
Edward Turner	University of Surrey, UK
Dennis Walter	Bremen University, Germany
David Williams	University of Surrey, UK

AVoCS 2009 Organizing Committee

Erwin R. Catesbeiana (Jr)	Swansea University, Wales, UK
Phillip James	Swansea University, Wales, UK
Temesghen Kahsai	Swansea University, Wales, UK
Liam O'Reilly	Swansea University, Wales, UK
Markus Roggenbach	Swansea University, Wales, UK

Contents

Invited Talks

Ulrich Berger
Portrait and CV 8
Proofs-as-Programs in Computable Analysis 9

Christoph Lüth
Portrait and CV 14
User Interfaces for Theorem Provers: Necessary Nuisance or Unexplored Potential? 15

Full Papers

Reynald Affeldt, David Nowak and Kiyoshi Yamada
Certifying Assembly with Formal Cryptographic Proofs: the Case of BBS 25

Étienne André, Laurent Fribourg, and Jeremy Sproston
An Extension of the Inverse Method to Probabilistic Timed Automata 41

Ulrich Berger and Sion Lloyd
A coinductive approach to verified exact real number computation 61

Loïc Besnard, Thierry Gautier, Matthieu Moy, Jean-Pierre Talpin, Kenneth Johnson and Florence Maraninchi
Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form 77

Neil C. C. Brown
Automatically Generating CSP Models for Communicating Haskell Processes 93

Nathaniel Charlton and Bernhard Reus
A decidable class of verification conditions for programs with higher order store 105

Holger Gast and Julia Trieflinger
High-level Proofs about Low-level Programs 123

Michael Huth, Nir Piterman and Huaxin Wang
A workbench for preprocessor design and evaluation: toward benchmarks for parity games 139

H. Palikareva, J. Ouaknine and A. W. Roscoe
Faster FDR Counterexample Generation Using SAT-Solving 155

Julio C. Peralta and Thierry Gautier
Towards SMV *Model Checking of* SIGNAL *(multi-clocked) Specifications* 171

F.P.M. Stappers and M.A. Reniers
Verification of safety requirements for program code using data abstraction 187

Beeta Vajar, Steve Schneider and Helen Treharne
Mobile CSP||B 205

Short Submissons

Anaheed Ayoub, Ayman Wahba, Ashraf Salem and Mohamed Sheirah
B Based Verification for Real Time Systems 225

Alan Bundy, Gudmund Grov and Cliff B. Jones
Learning from experts to aid the automation of proof search 229

Karim Kanso and Anton Setzer
Specifying Railway Interlocking Systems 233

Savas Konur, Ahmed Al Zahrani and Michael Fisher
Verification of a Message Delivery System using PRISM 237

Alexei Lisitsa
Reachability as deducibility, finite countermodels and verification 241

Sadouanouan Malo and Annie Choquet-Geniet
Analysis of critical scalable real-time systems by means of Petri nets 245

C.-H. L. Ong and N. Tzevelekos
Functional Reachability 249

Nikolaos Papanikolaou, Sadie Creese and Michael Goldsmith
Policy Refinement Checking 253

Shamim H. Ripon, Alice Miller and Alastair F. Donaldson
A Semantic Embedding of Promela-Lite in PVS 257

Monika Seisenberger
Program Verification via Extraction from Coinductive Proofs 261

Invited Talks

Ulrich Berger

Ulrich Berger is a Reader in Computer Science at Swansea University, Wales, UK. His research areas are Domain Theory, Proof Theory and Lambda Calculus. He received his PhD from the University of Munich, Germany, with a dissertation in Mathematics on "Total Objects and Sets in Domain Theory". His current research interests are Coinductive Definitions with applications to Program Extractions from Proofs in Constructive Analysis.

Proofs-as-Programs in Computable Analysis

Ulrich Berger

Swansea University, Wales, UK

Abstract: Since the work of Brouwer, Kolmogorov, Goedel, Kleene and many others we know that constructive proofs have computational meaning. In Computer Science this idea is known as the "proofs-as-programs paradigm" or "Curry-Howard correspondence". We present examples from computable analysis that show that this paradigm not only works in principle, but can be used to automatically synthesise practically relevant certified programs.

Keywords: Proof theory, program extraction, exact real number computation, coinduction

Besides the contributions to its intrinsic domains such as philosophy and the foundations of mathematics there are many applications of logic in areas outside logic. These applications mainly come from logical disciplines such as set theory, model theory, computability theory and modal logic and include, for example, in mathematics, proofs of the existence of certain algebraic structures, and, in computer science, methods for specifying and verifying computing systems and estimating their computational complexity. These applications are widely known, and they form a well-established part of theoretical and practical research, in particular in Computer Science.

Less known are extra-logical applications from *proof theory*, a branch of logic that has formal proofs as its main object of study. Classic results here are bounding informations about functions whose totality (i.e. termination) is proven in a certain formal system. For example, in the case of Peano Arithmetic the growth of the function is bounded by a transfinite extension of the Ackermann Function below the ordinal ε_0 (the limit of towers of ω exponentials) [Gen43, Wai72]. If one further restricts the induction scheme to purely existential formulas, then the function is even primitive recursive [Par72, Göd58]. Still, this is of little practical interest since primitive recursion goes far beyond feasible computability. The extraction of (from a computer science point of view) more relevant information, for example polynomial time or lower complexity, requires further (severe) restrictions of the proof rules [Bus86, CU93, Lei95, BNS00, OW05, ABHS04]. Other kinds of bounding information relevant in Approximation Theory, Functional Analysis and similar areas are obtained by Kohlenbach's monotone variant of Gödel's Functional Interpretation [Koh08].

A further large class of applications of proof theory in computer science can be summarised under the so called "proofs-as-programs paradigm" a.k.a. "Curry-Howard correspondence" which is the observation that –under certain conditions– formal proofs can be viewed and executed as programs. This correspondence is most direct for intuitionistic natural deduction (or Hilbert-style) systems that can immediately be viewed as dependently typed lambda-calculi (or combinatory logics) [HLS72, Tro73, vD80, GLT89] and hence as functional programming languages. Extensions of this correspondence to systems with classical logic have led to interesting tech-

niques such as programming with continuations and control operators [FF87]. and extensions of the lambda-calculus such as the lambda-mu-calculus [Par92].

In this note we want to draw the reader's attention to a very direct (and some my say bold) application of the proofs-as-program paradigm namely the extraction of practically useful programs from formal proofs. To this end it is necessary to remove from proofs all purely logical information and retain only the computationally relevant parts. This can be achieved by a proof-theoretic technique known as *realisability* that goes back to Kleene and Kreisel [Kle59, Kre59]. The realisability interpretation not only extracts an executable program, but also a specification of that program and a formal proof that the specification is fulfilled.

It would be an exaggeration to claim that program extraction is already being applied in practice, but some substantial case studies have been carried out indicating that the method is feasible and has the potential to become a viable method for safe software engineering in the future. We mention three of these case studies: The first is a fast higher-order algorithm for normalising simply typed lambda-terms that was extracted from Tait's normalisation proof [Tai67, Ber93] (the program had ben discovered earlier by Schwichtenberg [BS91] and is known as "normalisation-by-evaluation"). This case study was carried out independently in the proof assistants Coq, Isabelle and Minlog [BBLS06]. This normalisation program is unusual because it utilises higher types of any level to perform a purely syntactic task. The second example is concerend with Dickson's Lemma a result in infinitary combinatorics that is used e.g. in Gröbner Base theory. Dickson's lemma states that for fixed n the set of n-tuples of natural numbers is well-quasiordered (famous generalisations of this are Higman's Lemma, Kruskal's Theorem and the Graph Minor Theorem). The extracted program computes for every infinite sequence of n-tuples a pair of indices $i < j$ such that the i-th tuple is pointwise $\leq$ the j-th [BSS01]. The interesing aspect of this example is that the program is extracted from the classical i.e. non-constructive proof by Nash-Williams [NW63] using a version of Friedman's A-translation [Fri78] that translates classical into constructive proofs. Finally, we mention the extraction of programs from the Intermediate Value Theorem and the Inverse Function Theorem for continous real functions with a positive lower bound on the slope [Sch06]. These are probable the first non-trivial examples of program extraction in Computable Analysis.

While the last of the three case studies above is based on the Cauchy representation of real numbers, our recent work on program extaction is based on the signed digit representation. More precisely, we use inductive and coinductive defintions to characterise unifomly continuous functions in such a way that one can extract an implementation of such functions by non-wellfounded trees that act as transformers of signed digit streams. A suitable adaptation of realisability to this setting is described in [Ber09]. We have extracted from constructive proofs programs that compute high iterations of the logistic map, integrals of continuous functions, the constant π, and functions defined power series.

The two last examples make use of a more general setting where the set of signed digits $\mathrm{SD} := \{0,1,-1\}$, which is to be thought of as the set of representations of the contractions $\mathrm{av}_d(x) := (d+x)/2$ ($d \in \mathrm{SD}$), is replaced by an arbitrary set D of endomaps on a set X. Large parts of our theory can be developed for arbitrary such structures (X,D) which we call *digit spaces*. This greater generality not only leads to clearer proofs but also to new algorithms, the last two case studies mentioned above being examples.

The programs in the latter case studied have been extracted "by hand" from proofs in the

theory of digit spaces which are "nearly formal". The implementation of the theory of digit spaces using a suitable proof assistant (for example, Minlog or Coq) is a matter of future work.

The method of program extraction from proofs can be viewed as a consistent and rather radical continuation of methods of program development, advocated by Dijkstra and others [Dij97, Gri81, DJ78] where programs are correct "by construction". Whether or not these methods will eventually be accepted and used in practice remains a question of the future. The experimental results obtained so far are encouraging, however.

Bibliography

[ABHS04] K. Aehlig, U. Berger, M. Hofmann, H. Schwichtenberg. An arithmetic for non-size-increasing polynomial-time computation. *Theor. Comput. Sci.* 318:3–27, 2004.

[BBLS06] U. Berger, S. Berghofer, P. Letouzey, H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica* 82:25–49, 2006.

[Ber93] U. Berger. Total Sets and Objects in Domain Theory. *Ann. Pure Appl. Logic* 60:91–117, 1993.

[Ber09] U. Berger. Realisability for induction and coinduction. 2009. To appear: Proceedings of Computability and Complexity in Analysis (CCA), Ljubljana, Slovenia, 18-22 August, 2009.

[BNS00] S. Bellantoni, K.-H. Niggl, H. Schwichtenberg. Higher Type Recursion, Ramification and Polynomial Time. *Ann. Pure Appl. Logic* 104:17–30, 2000.

[BS91] U. Berger, H. Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In Vemuri (ed.), *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*. Pp. 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[BSS01] U. Berger, H. Schwichtenberg, M. Seisenberger. The Warshall Algorithm and Dickson's Lemma: Two Examples of Realistic Program Extraction. *J. Autom. Reasoning* 26(2):205–221, 2001.

[Bus86] S. Buss. *Bounded Arithmetic*. Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1986.

[CU93] S. Cook, A. Urquhart. Functional Interpretations of Feasibly Constructive Arithmetic. *Ann. Pure Appl. Logic* 63:103–200, 1993.

[vD80] D. van Dalen. *Logic and Structure*. Springer–Verlag, Berlin, 1980.

[Dij97] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[DJ78] B. Dines, C. Jones. *The Vienna Development Method: The Meta-Language*. LNCS 61. Springer, Berlin, Heidelberg, New York, 1978.

[FF87] M. Felleisen, D. P. Friedman. Control Operators, the SECD–Machine, and the λ–Calculus. In Wirsing (ed.), *Formal Description of Programming Concepts — III*. Pp. 193–219. Elsevier (North–Holland), Amsterdam, 1987.

[Fri78] H. Friedman. Classically and intuitionistically provably recursive functions. In Scott and Müller (eds.), *Higher Set Theory, Lecture Notes in Mathematics*. Volume 669, pp. 21–28. Springer, 1978.

[Gen43] G. Gentzen. Beweisbarkeit und Unbeweisbarkeit von Anfangsfällen der transfiniten Induktion in der reinen Zahlentheorie. *Mathematische Annalen* 119:140–161, 1943.

[GLT89] J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Göd58] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica* 12:280–287, 1958.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[HLS72] J. Hindley, B. Lercher, J. Seldin. *Introduction to Combinatory Logic*. London Mathematical Society Lecture Notes Series 7. Cambridge University Press, 1972.

[Kle59] S. C. Kleene. Countable functionals. In Heyting (ed.), *Constructivity in Mathematics*. Pp. 81–100. North–Holland, 1959.

[Koh08] U. Kohlenbach. *Proof Interpretations and their Use in Mathematics*. Springer Monographs in Mathematics. Springer, 2008.

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics*, pp. 101–128, 1959.

[Lei95] D. Leivant. Ramified Recurrence and Computational Complexity I: Word Recurrence and Poly–time. In Clote and Remmel (eds.), *Feasible Mathematics II*. Pp. 320–343. Birkhäuser, Boston, 1995.

[NW63] C. Nash-Williams. On well-quasi-ordering finite trees. *Proc. Cambridge Phil. Soc.* 59:833–835, 1963.

[OW05] G. Ostrin, S. Wainer. Elementary Arithmetic. *Ann. Pure Appl. Logic* 133:275–292, 2005.

[Par72] C. Parsons. On *n*-quantifier induction. *Jour. Symb. Logic* 37:466–482, 1972.

[Par92] M. Parigot. $\lambda\mu$–calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of Logic Programming and Automatic Reasoning, St. Petersburg*. LNCS 624, pp. 190–201. Springer, 1992.

[Sch06] H. Schwichtenberg. Inverting monotone functions in constructive analysis. In Beckmann et al. (eds.), *CiE 2006: Logical Approaches to Computational Barriers*. LNCS 3988, pp. 490–504. Springer, 2006.

[Tai67] W. Tait. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32(2):198–212, 1967.

[Tro73] A. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Lecture Notes in Mathematics 344. Springer, 1973.

[Wai72] S. S. Wainer. Ordinal recursion, and a refinement of the extended Grzegorcyk hierarchy. *Jour. Symb. Logic* 37:281–292, 1972.

Christoph Lüth

Christoph Lüth is a senior researcher and vice director of the safe and secure cognitive systems group at the German Research Centre for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) in Bremen, Germany.

His research covers the whole area of advanced software development, from theoretical foundations as found in category theory to the development of tools to construct or verify software, and applications in practical areas such as robotics. The overall theme here is how to reliably construct correct software.

He holds a PhD from the University of Edinburgh, and a Habilitation from the University of Bremen, where he has been working as a lecturer (associate professor) prior to joining DFKI at the start of 2006. He has authored or co-authored over thirty peer-reviewed papers, and is currently the principal investigator in a research project funded by the federal ministry of education and research (BMBF) investigating provably safe control of automous mobile robots.

User Interfaces for Theorem Provers: Necessary Nuisance or Unexplored Potential?

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz, Bremen

Abstract: This note considers the design of user interfaces for interactive theorem provers. The basic rules of interface design are reviewed, and how they are applicable to theorem provers, leading to considerations about the particular challenges of interface design for theorem provers. A short overview and classification of existing interfaces is given, followed by suggestions of possible future work in the area.

Keywords: user interfaces, theorem provers

1 Introduction

Theorem provers need to be interactive, and interactive theorem provers need user interfaces. The first part of this statement may sound controversial, but even fully automatic theorem provers need a way to state the proposition to be proven, and the hard facts of logic mean that any meaningful proofs, be it about program verification, formalised mathematics, or any other application area, will have to have be conducted with human interaction. So user interfaces are a necessary nuisance, but do they offer more potential?

For most interactive theorem provers, user interfaces have been something of an afterthought in the beginning — understandably so, as developing the core technology was enough of a challenge. With the advances of this technology over the recent years, theorem proving has come of age. The use of interactive proof spread beyond its previous confines, from case studies to real applications (e.g. in mathematics, software or hardware verification), and with new users the need for better interfaces arises.

In this note, we consider *interactive* theorem provers (the most famous examples of which are, in particular order, Coq, HOL, HOL light, Isabelle, and PVS), which read *proof scripts* containing definitions, declarations, theorems and prover-specific commands. These proof scripts are the central artefacts under construction; supporting the user to interactively construct them should be the main purpose of the user interface. Even though phrased in terms of theorem proving, the discussion also pertains to interactive formal method tools; we come back to this in the conclusions.

We first consider criteria for a good user interface for theorem provers, and then review existing interfaces and their strengths and weaknesses. From this, we point out some directions of future research, and conclude with the major challenges and a résumé.

(i) Strive for consistency.
(ii) Cater to universal usability.
(iii) Offer informative feedback.
(iv) Design dialogs to yield closure.
(v) Prevent errors.
(vi) Permit easy reversal of actions.
(vii) Support internal locus of control.
(viii) Reduce short-term memory load.

Figure 1: The 'Eight Golden Rules' of interface design. Taken from [SP09, p. 88f].

2 What makes a Good User Interface?

2.1 General Criteria

Figure 1 shows the 'golden rules' of interface design according to Shneiderman [SP09] (other authors give similar guidelines). They are all relevant for interactive theorem proving, but some rules particularly so: the sixth, because interactive theorem by its very nature is an explorative process, so it is important to be able to try and undo proof steps; the eighth, alluding to the seven (plus or minus two) chunks of information that can be held in short-term memory, because theorem provers can actually offer a lot of information to the user, and the problem is to not overwhelm the user; and the second, because users will range from experts who know exactly what they want and how to achieve it and might prefer a programmable command line interface, to complete novices who need every help they can get, and prefer syntax-free interface elements such as menus.

On the other hand, the seventh rule should be taken with a grain of salt. It means that it should be easy to get the system to do what the user wants to achieve, which in a theorem prover means proving propositions. Unfortunately, keeping the user from proving wrong propositions is the core of interactive theorem proving, and the resulting frustrations are par for the course (and cannot be blamed on the interface). Showing *why* a particular action does not work, on the other hand, is an important task of the interface, so the rule should be read in this sense.

2.2 Challenges in User Interfaces for Theorem Provers

Theorem provers are special programs, and designing a good interface for them offers special challenges. The first difficulty is that theorems or proofs are abstract in the sense that they have no physical counterpart. Hence, *syntax* plays a central role in theorem proving (and mathematics), since syntax is what we manipulate. The capability to read and write proofs in a notation which is close to what users are used to from text books cannot be overstated, because it eases the cognitive load on the user considerably.

Secondly, theorem proving is very hard, and proof scripts are very abstract in the sense that they condense much information; they cannot be manipulated as conveniently as e.g. source code. The problem here is the high degree of interdependency, which tends to make proof scripts brittle, so changing them in one place may lead to unexpected failure elsewhere, which makes

changing proofs very frustrating.

Thirdly, theorem provers potentially offer a lot of information: the proof state can become very large, the amount of rules, theorems, proof procedures known to the system can run into thousands, etc. It is important not to overwhelm the user, but it is even as important (and more challenging to implement) to allow the user to query the system interactively, preferably at different levels of abstraction, depending on the user's proficiency.

The most important consequence of these considerations is that user interface and theorem prover need to interact closely, with control flow going in both directions; interface design for theorem prover is more than 'bolting a bit of Tcl/Tk onto a text-command-driven existing prover in an afternoon's work' [BS98].

3 A Review of User Interfaces Past and Present

3.1 The Early Days

In the early days, interactive theorem provers were used from the command line. Users wrote a proof script which they fed to the prover, and the prover would check it; the interaction was in batch mode, very much like a with a compiler. Although today this modus operandi would be considered unproductive, it was standard practice back then. Moreover, interactive theorem proving is merely a niche, and subsequently resources to develop user interfaces have always been scarce (comprised mostly of postgraduate students struggling to produce a thesis under the limitations of the prevailing user interface, and gratefully finding some justifiable diversion from their thesis work).

3.2 Emacs and Proof General

Under these limitations, the Emacs editor, which allows for comfortable and flexible customisation using the Lisp dialect it is written in, offered an excellent platform, and soon specialised Emacs modes for many of the popular provers appeared. After a while, it became clear that many provers shared a similar interaction mode, and maintaining each of them separately was an unnecessary burden. The Proof General project [Asp00] in Edinburgh consolidates the different Emacs interfaces for Isabelle, Lego and others into one Emacs package which can be instantiated to the different provers.

Proof General found widespread use and is the most popular interface implementing the idea of *script management* introduced in [BT98], where the proof script is treated as a sequence of commands, which are processed in a linear fashion. This divides the script in three regions, one of which has already been processed, one which is currently being processed, and one of which is unprocessed. Once a region has been processed, it can not be edited anymore. A simple undo function allows the user to go back in the proof. This idea is strikingly simple and powerful; it is cheap to implement on the prover's side, and on the other hand offers a flexible way to add more functionality in the user interface (e.g. a 'go to here' button, which performs forward or backward steps as required).

3.3 Integrated Development Environments

An integrated development environment (IDE) offers a tight integration of source code editor, compiler, debugger, documentation browser, and other tools. SmallTalk was the first language to come with an IDE, which became popular with Borland's Turbo-Pascal. With the similarities between theorem proving and software development (in both, the object of interest — source code and proof script respectively — is processed by an external tool — compiler or theorem prover), it seems tempting to construct an IDE for theorem proving [TBK92]. Early attempts include CtCoq and PCoq [ABPR01]. These attempts have been hampered by the fact that the integration between a theorem prover and its interface needs to be far closer than between a compiler and a source-code editor, and that in particular maintaining an IDE is a substantial task — many of these efforts fall out of use because the underlying prover changes and is no longer compatible with the interface. Recently, powerful interface toolkits made it easier to create IDEs, such as the CoqIDE created using GTK+, but this is still the exception rather than the rule.

3.4 Graphical User Interfaces

Graphical user interfaces (GUIs) entered the scene as early as the 70s with the Xerox Star system. The methodology behind graphical user interfaces is *direct interaction*: all objects of interest are represented continuously and graphically, preferably using an understandable metaphor, and can be manipulated with syntax-free operations on this representation, such as pointing at them, moving them, or causing interaction by dropping them onto other objects.

Finding such a metaphor for theorem provers is a challenge, since the objects in a theorem prover are abstract, and it is far from clear how their manipulation can be modelled by intuitive gestures, although attempts have been made [LW99]. The Jape system was a pioneering effort [BS99]; it was designed to be a 'quiet interface', meaning it would only show as little as needed and not as much as possible (which as pointed out above is good interface design practice), and uses gestures to select proof steps.

3.5 Document-Centered Approaches

The PVS system has developed a closer interaction model with the Emacs editor then the other systems mentioned in Sect. 3.2. The user is essentially editing an interactive document in the editor's buffer, with the prover checking the semantic integrity in the background. This is the so-called *document-centered* approach, where the focus of attention is the proof script itself, and how to edit it, rather than it being processed by a prover. It works best with a style of proof scripts which is not a simple sequence of state-affecting prover commands, but where the proof script represents the proof itself, e.g. by stating a sequence of transformations or intermediate goals. The Mizar prover pioneered this approach [T+73], and Isar brought it to the Isabelle system [Wen99]. Taking this idea one step further is the Plato system [WAB06], which uses the Texmacs editor to provide WYSIWYG editing of mathematical documents in a LaTeX-like language with high quality typesetting, while the proofs are checked by the Omega-prover in the background.

4 The Future of User Interfaces

What have the interfaces introduced in the last chapter achieved? Without wishing to denigrate the efforts of the researchers involved, there is still a lot of room for improvement. What we can take from the existing interfaces is that as Proof General shows, it is good to be generic. Hardly any theorem prover has a large enough developer base to develop its own interface, but by sharing the effort across different provers we can achieve something. Genericity is also good because it helps to make the connection between interface and prover clear; e.g. the interaction protocol for Proof General was made explicit in the PGIP protocol [ALW07]. It is also important to note that the success of a prover hinges mainly on its expressiveness and proof support; in the past, users have always preferred a powerful prover with an Emacs interface over a less powerful prover with a slick GUI, even if the latter is easier to use. The aim must be, then, to provide existing powerful provers with better interfaces.

4.1 Modern IDEs

Early attempts to develop IDEs for theorem proving have been mentioned above. With modern IDEs such as Eclipse and NetBeans which are specifically designed to be generic, the situation has improved, and it is tempting to instantiate e.g. Eclipse as a theorem proving interface [ALWF06]. However, Eclipse is not exactly light-weight, and a major disadvantage of most IDEs is that they do not support mathematical notation well.

Particularly appealing in Eclipse is its incremental document processing. That is, there is no explicit 'process this document' step, rather the prover (or compiler) continuously processes as much of the document as possible in the background, flagging up errors as they occur. This asynchronous mode of interaction makes good use of the time the user spends thinking, increasing overall responsiveness of the system.

4.2 Emerging Technologies

The most drastic change in interface technology over the last years has possibly been the rise of web-based technologies. The technique known as AJAX (asynchronous Java script and XML) has taken web-based interfaces from filling in forms to fully interactive graphical user interfaces, and in future the distinction between local (desktop-based) and remote (web-based) interfaces will probably be blurred even further. These technologies can play a rôle in theorem proving too, as they allow easy cross-platform access to a theorem prover without having to install it locally, often a daunting task for the novice. An impressive early attempt here is Kaliszyk's ProofWeb [Kal07].

4.3 Interaction Models

There have been various attempts to adapt more intuitive interaction models like gestures into theorem proving interfaces, like in Jape or Coq ('proof-by-pointing' [BKS97]). It seems tempting to allow the user to rearrange formulae by drag-and-drop, going beyond what pen-and-paper mathematics allows us to do. However, this has to be reconciliated with the fact that the main artefact of a theorem prover is the proof script; a proof consisting of a series of gestures is not

really useful. Thus, gestures should be seen as a way to create proof scripts. Users indicate that they wish to perform induction on x, or exchange the two arguments of $+$ (and this can be done either via drag-and-drop gestures, a menu button, or even more exotic means), the prover returns a new proof script fragment, which the interface inserts into the proof script. The challenge is to provide a uniform interaction protocol which works reliably across different provers (a first attempt has been made in [ALW06]).

4.4 Foundations

Interfaces have mostly been seen in technological terms. This is understandable, because technology is what delivers to the user, but the theoretical foundations of user interfaces have not received much attention. An exception is Denney's work [DPT05], which introduced the notion of a hierarchical proofs, and operations such as zooming into a proof, on a purely semantics-free level. This allows interfaces to implement operations on this level, separating the purely syntactic manipulation which can be done in the interface from the semantic manipulations in the theorem prover.

5 Conclusions

We have highlighted the challenges in constructing interfaces for theorem provers, reviewed existing interfaces, and pointed out some directions of future research. All of this is necessarily subjective, so the author is grateful for any omissions pointed out to him. The discussion here has been phrased in terms of interactive theorem provers, but applies equally well to formal methods tools; the key difference between formal method tools and theorem provers is that because formal method tools typically have a more singular purpose (e.g. proofs in a particular notation or of particular properties), users and their level of proficiency will be less diverse, but the points about consistent notation and genericity are equally valid.

As a closing summary, the key technical challenges in the author's estimate are the comprehensive support of mathematical notation (maybe standard vector graphics formats such as SVG can offer a solution here), and a clear standard protocol for theorem provers to interact with user interfaces. PGIP was a first start in this direction, but it possibly is oriented too much towards script management; a recent new version [AALW09] aims to rectify this shortcoming.

The overall challenge in user interfaces is to leverage the underlying technology to an extent which makes it *easier* to do proofs in a computer than with pen and paper. Presently, this is not the case. Theorem provers tend to get in the way more often than they are helpful, and even though that is in part their duty as proof checkers, the preferable rôle model of a theorem prover should be that of a helpful co-author gently pointing out errors and suggesting improvements, rather than a stubborn civil servant refusing to accept the blindingly obvious because of some formality. This in good part an interface issue, and hence the author's answer to the initial question is that there is definitely unexplored potential, waiting to be developed by enterprising minds.

Acknowledgements: Research in part supported by the German Research Agency (DFG) under grant LU 707-2/2.

Bibliography

[AALW09] D. Aspinall, S. Autexier, C. Lüth, M. Wagner. Towards Merging PlatΩ and PGIP. In *Proc. 8th International Workshop on User Interfaces for Theorem Provers (UITP 2008)*. Electronic Notes in Theoretical Computer Science 226, pp. 3– 21. Elsevier Science, 2009.

[ABPR01] A. Amerkad, Y. Bertot, L. Pottier, L. Rideau. Mathematics and Proof Presentation in PCOQ. In *Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01), Sienna, Italy*. 2001. also available as INRIA RR-4313.

[ALW06] D. Aspinall, C. Lüth, B. Wolff. Assisted Proof Document Authoring. In Kohlhase (ed.), *Mathematical Knowledge Management MKM 2005*. Lecture Notes in Artificial Intelligence 3863, pp. 65– 80. Springer, 2006.

[ALW07] D. Aspinall, C. Lüth, D. Winterstein. A Framework for Interactive Proof. In *Mathematical Knowledge Management MKM 2007*. LNAI 4573, pp. 161– 175. Springer, 2007.

[ALWF06] D. Aspinall, C. Lüth, D. Winterstein, A. Fayyaz. Proof General in Eclipse. In *Eclipse Technology eXchange ETX'06*. ACM Press, 2006.

[Asp00] D. Aspinall. Proof General: A Generic Tool for Proof Development. In Graf and Schwartzbach (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science 1785, pp. 38–42. Springer, 2000.

[BKS97] Y. Bertot, T. Kleymann, D. Sequeira. Implementing Proof by Pointing without a Structure Editor. Technical report ECS-LFCS-97-368, University of Edinburgh, 1997. Also published as Rapport de recherche de l'INRIA Sophia Antipolis RR-3286.

[BS98] R. Bornat, B. Sufrin. Using gestures to disambiguate unification. In *User Interfaces for Theorem Provers UITP'98*. 1998.

[BS99] R. Bornat, B. Sufrin. A minimal graphical user interface for the Jape proof calculator. *Formal Aspects of Computing* 11(3):244– 271, 1999.

[BT98] Y. Bertot, L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation* 25(7):161–194, Feb. 1998.

[DPT05] E. Denney, J. Power, K. Tourlas. Hiproofs: A hierarchical notion of proof tree. In *Proceedings of Mathematical Foundations of Programing Semantics (MFPS)*. Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2005.

[Kal07] C. Kaliszyk. Web Interfaces for Proof Assistants. In Autexier and Benzmüller (eds.), *Proc. User Interfaces for Theorem Provers (UITP'06)*. ENTCS 174(2), pp. 49–61. 2007.

[LW99] C. Lüth, B. Wolff. Functional Design and Implementation of Graphical User Interfaces for Theorem Provers. *Journal of Functional Programming* 9(2):167– 189, Mar. 1999.

[SP09] B. Shneiderman, C. Plaisant. *Designing the User Interface*. Addison-Wesley, 5th edition, 2009.

[T+73] A. Trybulec et al. The Mizar Project. 1973. See web page hosted at http://mizar.org, University of Bialystok, Poland.

[TBK92] L. Théry, Y. Bertot, G. Kahn. Real theorem provers deserve real user-interfaces. *SIGSOFT Softw. Eng. Notes* 17(5):120–129, 1992.

[WAB06] M. Wagner, S. Autexier, C. Benzmüller. PLATΩ: A Mediator between Text-Editors and Proof Assistance Systems. In Autexier and Benzmüller (eds.), *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*. ENTCS. Elsevier, 2006.

[Wen99] M. Wenzel. Isar — a Generic Interpretative Approach to Readable Formal Proof Documents. In Bertot et al. (eds.), *Theorem Proving in Higher Order Logics TPHOLs'99*. Lecture Notes in Computer Science 1690, pp. 167– 184. Springer, 1999.

Full Papers

Certifying Assembly with Formal Cryptographic Proofs: the Case of BBS

Reynald Affeldt, David Nowak and Kiyoshi Yamada

Research Center for Information Security, AIST, Japan

Abstract: With today's dissemination of embedded systems manipulating sensitive data, it has become important to equip low-level programs with strong security guarantees. Unfortunately, security proofs as done by cryptographers are about algorithms, not about concrete implementations running on hardware. In this paper, we show how to perform security proofs to guarantee the security of assembly language implementations of cryptographic primitives. Our approach is based on a framework in the Coq proof assistant that integrates correctness proofs of assembly programs with game-playing proofs of provable security. We demonstrate the usability of our approach using the Blum-Blum-Shub (BBS) pseudorandom number generator, for which a MIPS implementation for smartcards is shown cryptographically secure.

Keywords: Hoare logic, Assembly language, Coq, PRNG, Provable security

1 Introduction

With today's dissemination of embedded systems manipulating sensitive data, it has become important to equip low-level programs with strong security guarantees. However, despite the fact that most security claims implicitly assume correct implementation of cryptography, this assumption is never formally enforced in practice. The main problem of formal verification of embedded cryptographic software is that, in the current state of research, formal verification remains a major undertaking:

(a) Most cryptographic primitives rely on number theory and their pervasive usage calls for efficient implementations. As a result, we face many advanced algorithms with low-level implementations in assembly language. This already makes formal proof technically difficult.

(b) Security guarantees about cryptographic primitives is the matter of *cryptographic proofs*, as practiced by cryptographers. In essence, these proofs aim at showing the security of cryptographic primitives by reduction to computational assumptions. Formal proofs of such reductions also involve probability theory or group theory.

In addition, formal verification of embedded cryptographic software is even more challenging in that it requires a formal integration of (a) and (b). In fact, to the best of our knowledge, no such integration has ever been attempted so far.

In this paper, we address the issue of formal verification of cryptographic assembly code with cryptographic proofs. As pointed out above, formal verification of cryptographic assembly code and formal verification of cryptographic proofs are not the same matter, even though both deals with cryptography. As an evidence of this mismatch, one can think of a cryptographic function such as encryption: its security proof typically relies on a high-level mathematical description,

but when laid down in terms of assembly code such a function exhibits restrictions due to the choice of implementation. We are therefore essentially concerned about the integration of these two kinds of formal proofs. We do not question here the theoretical feasibility of such an integration; rather, we investigate its practical aspects. Indeed, various frameworks for formal verification of cryptography using proof assistants based on proof theory already exist ([AM06, MG07] for cryptographic assembly code, [ATM07, BBU08, BGZ09, Now07] for cryptographic proofs), but it is not clear how to connect them in practice. Whatever connection is to be provided, it has to be developed in a clear way, both understandable by cryptographers and implementers, and in a reusable fashion, so that new verification efforts can build upon previous ones.

Our main contribution is to propose a concrete approach, supported by a reusable formal framework on top of the Coq proof assistant, for verification of assembly code together with cryptographic proofs. As a concrete evidence of usability, we formally verify a pseudorandom number generator written in assembly for smartcards with a proof of unpredictability. This choice of application is not gratuitous: this is the first step before verifying more cryptographic primitives, since many of them actually rely on pseudorandom number generation. To achieve our goal, we integrate two existing frameworks: one for formal verification of assembly code, and another for formal verification of cryptographic primitives. More precisely, our technical contributions consist in the following:

• We propose an integration in terms of *game-playing* [Sho04], a popular setting to represent cryptographic proofs. We introduce a new kind of game transformation to serve as a bridge between assembly code and algorithms as dealt with by cryptographers. This allows for a clear integration, that paves the way for a modular framework, understandable by both cryptographers and implementers.

• We extend the formal framework for assembly code of [AM06] to connect with the formal framework for cryptographic proofs of [Now07]. Various technical extensions are called for, that range from the natural issue of encoding mathematical objects such as arbitrarily-large integers into computer memory, to technical issues such as composition of assembly snippets to achieve verification of large programs. All in all, it turns out that it is utterly important to provide efficient ways to deal with low-level details induced by programs being written in assembly. Here, we explain in particular how we deal with arbitrary jumps in assembly. Concretely, we provide a formalization of the proof-carrying code framework of [SU07], that allows us to verify assembly with jumps through standard Hoare logics proofs.

• We provide the first assembly program for a pseudorandom number generator that is formally verified with a cryptographic proof. The generator in question is the Blum-Blum-Shub pseudorandom number generator [BBS86] that we implement in the SmartMIPS smartcard assembly.

Alternative Approaches and Related Work Our approach is oriented towards practical application, and this goal includes formal verification of hand-written assembly. For this purpose, extension of [AM06] is appropriate because it already provides much material for reasonably short proof scripts. Under less constraints about the target code, the alternative approach of proof-producing compilation could be considered. Of course, a compiler such as the one of [MSG09] needs to be extended with custom support for cryptography to achieve reasonable performance. Still, an early application of ideas of proof-producing compilation to cryptographic functions shows that short proof scripts and compact assembly are difficult to reconcile [MG07]. Over-

coming these difficulties is still not enough for our overall goal, for HOL (the proof assistant used in [MSG09] and [MG07]) lacking a framework for formal cryptographic proofs.

The two existing frameworks ([AM06] and [Now07]) that we integrate in this paper turn out to be a good fit for they favoring shallow encodings. On the one hand, shallow encoding is used in [AM06] to encode Hoare logic assertions, and on the other hand, it is used in [Now07] to represent games. Therefore, algorithms written as Coq functions can simply appear in Hoare logic assertions, making for an easy integration. In contrast, games in [BBU08, BGZ09] are represented as deep-encoded code. In addition, our use-case directly relies on properties of arithmetic (including an encoding of the quadratic residuosity problem) an originality of [Now08].

Outline In Sect. 2, we introduce the BBS algorithm and provide an assembly implementation. In Sect. 3, we explain how we integrate formally proofs of functional correctness for assembly code with game-based cryptographic proofs. In Sect. 4, we explain our formalization of the proof-carrying code framework of [SU07], that facilitates formal proof of functional correctness of assembly code. In Sect. 5, we explain the formal proof of functional correctness of BBS and the lemmas relevant to the integration with its cryptographic proof. In Sect. 6, we comment on technical aspects of the Coq formalization. We conclude and comment on future work in Sect. 7.

2 The BBS Pseudorandom Number Generator

2.1 The BBS Algorithm

The Blum-Blum-Shub pseudorandom number generator [BBS86] (hereafter, BBS) exploits the *quadratic residuosity problem*. This problem is to decide whether integers have square roots in modular arithmetic. This is believed to be intractable for multiplicative group of integers modulo m where m is the product of two distinct odd primes. BBS exploits the quadratic residuosity problem in the particular case of m being a Blum integer, i.e., the product of two distinct odd primes congruent to 3 modulo 4.

Here follows an implementation of BBS as a Coq function. It performs iteratively squaring modulo and outputs the result of parity tests. The input is the desired number of pseudorandom bits (*len*) and a random seed (*seed*) for initialization. $\mathbb{Z}_m^*$ is the multiplicative group of integers modulo m and QR_m is the set of quadratic residues modulo m.

$$\begin{array}{l}\mathsf{bbs}(len \in \mathbb{N}, seed \in \mathbb{Z}_m^*) =_{def} \mathsf{bbs\_rec}(len, seed^2)\\ \mathsf{bbs\_rec}(len \in \mathbb{N}, x \in QR_m) =_{def} \textbf{match } len \textbf{ with}\\ \quad \mid 0 \Rightarrow []\\ \quad \mid len'+1 \Rightarrow \mathsf{parity}(x) :: \mathsf{bbs\_rec}(len', x^2)\\ \quad \textbf{end}\end{array}$$

BBS is one of the rare pseudorandom number generators that is *cryptographically secure*, i.e., it passes all polynomial-time statistical tests (no polynomial-time algorithm can distinguish between an output sequence of the generator and a truly random sequence). This strong property is not required of most applications of pseudorandom numbers, except cryptography. In practice, BBS can be proved *left-unpredictable* (hereafter, "unpredictable") under the assumption that the quadratic residuosity problem is intractable (this is equivalent to prove that BBS passes all polynomial-time statistical tests [Yao82].)

2.2 Implementation of BBS in Assembly

The assembly code bbs_asm in Fig. 1 implements BBS. It is written with MIPS instructions (actually, we use SmartMIPS, a superset of MIPS32 with additional instructions for smart-cards [Mips]). It consists of a loop with a nested loop. Each iteration of the nested loop produces

```
bbs_asm =def
0:    addiu i gpr_zero 0_16          (* init counter for outer loop *)
1:    addiu L l 0_16                 (* init pointer to result *)
2:    beq i n 240                    (* repeat n times *)
3:      addiu j gpr_zero 0_16        (* init counter for inner loop *)
4:      addiu w gpr_zero 0_16 (* init word of temporary storage *)
5:      beq j thirtytwo 236          (* repeat 32 times *)
6:        mul_mod k x x m ...        (* compute X^2 (mod M) *)
230:      lw w_ 0_16 x               (* load least significant word *)
231:      andi w_ w_ 1_16            (* extract parity bit *)
232:      sllv w_ w_ j               (* shift parity bit to jth position *)
233:      cmd_or w w w_ (* store parity bit in temporary storage *)
234:      addiu j j 1_16             (* increment inner loop counter *)
235:      jmp 5                      (* end of the inner loop *)
236:    sw w 0_16 L        (* store the last 32 parity bits in memory *)
237:    addiu L L 4_16               (* increment pointer to result *)
238:    addiu i i 1_16               (* increment outer loop counter *)
239:    jmp 2                        (* end of the outer loop *)
240:
```

Figure 1: The Blum-Blum-Shub pseudorandom number generator in assembly

one word of pseudorandom bits by performing a square modulo, extracting the parity bit (of the least significant word), and storing this bit in a temporary word of storage in an appropriate position using bitwise operations. These temporary words of storage are then stored in memory contiguously by the outer loop so as to produce a pseudorandom number. The names of registers (in italic font) are parameters; only the null register `gpr_zero` is hardwired in the program. Magic numbers are indexed with their length in bits (e.g., 0_{16} stands for 0 represented as a half-word). mul_mod is an inlined assembly program explained in the next section.

2.3 Implementation of Modular Multiplication in Assembly

We implement multi-precision square modulo using the Montgomery multiplication [Mon85]. This is not the fastest way to implement multi-precision square modulo but, still, this is reasonable: like the natural multi-precision multiplication/division, it has a quadratic complexity. Moreover, we already have a formal proof for an optimized version of the Montgomery multiplication [AM06], whereas, to the best of our knowledge, such a formal proof for multi-precision division does not exist yet.

Using Montgomery, modular multiplication is performed as follows. Given three k-word integers M, X, Y, the Montgomery multiplication computes a $k+1$-word integer Z such that

$\beta^k Z = X.Y \pmod M$ and $Z < 2M$ ($\beta = 2^{32}$). This is almost a multiplication modulo except for the parasite value β^k and because $Z \not< M$ in general. To turn it into a genuine multiplication modulo, one needs (1) an additional subtraction to reduce Z by M when necessary and (2) two passes to eliminate the parasite value. The second pass requires as an additional input a k-word $A = \beta^{2k} \pmod M$; given Z such that $\beta^k Z = X.Y \pmod M$, it suffices to compute Z' such that $\beta^k Z' = Z.A \pmod M$: if M is odd (this is generally the case for cryptographic applications), one obtains as desired $Z' = X.Y \pmod M$.

The assembly code mont_mul_strict_init in Fig. 2 implements the Montgomery multiplication extended with comparison and subtraction. It makes use of the functions montgomery (the

```
mont_mul_strict_init =def
6:    multi_zero ext k Z z                          (* output initialization *)
13:   mflhxu gpr_zero                               (* multiplier inialization *)
14:   mthi gpr_zero
15:   mtlo gpr_zero
16:   montgomery k alpha x y z m one ext int X Y M Z quot C t s
54:   beq C gpr_zero 81                  (* is the output k+1-word long? *)
55:     addiu t t 4_16
56:     sw C 0_16 t
57:     addiu ext k 1_16
58:     multisub ext one z m z M int quot C Z X Y X
80:     jmp 118
81:     multi_lt_prg k z m X Y int ext Z M
93:     beq int gpr_zero 96      (* is the output bigger than the modulus? *)
94:       skip
95:       jmp 118
96:       multisub k one z m z ext int quot C Z X Y X
118:
mul_mod =def
6:    mont_mul_strict_init k alpha x x y m one ext int X B2K Y M quot C t s
118:  mont_mul_strict_init k alpha y b2k x m one ext int X B2K Y M quot C t s
320:
```

Figure 2: The Montgomery multiplication extended with comparison and subtraction

function certified in [AM06]), (in-place) subtraction multi_sub (derived from [AM06]), multi-precision comparison multi_lt and an initialization function multi_zero (see [Code] for the details). The assembly code mul_mod in Fig. 2 perform a square modulo by using twice mont_mul_strict_init, provided we assume that the register $b2k$ points to the pre-computed k-word $A = \beta^{2k} \pmod M$.

3 Game-based Proofs for Assembly

Cryptographic proofs usually apply to algorithms without any consideration for implementation. In order to prove unpredictability directly on assembly code, we propose to lift a standard definition borrowed from game-playing [Sho04]. Game-playing is a methodology to write cryptographic proofs that are easier to verify; it lends itself well to formalization [ATM07, BBU08,

BGZ09, Now07]. A security property is modeled as a *game* (a program) to be solved by an attacker, the latter being modeled as some probabilistic procedure. A cryptographic proof consists in showing that any attacker has only little advantage over a random player, by (1) stating the security property for the cryptographic primitive to be verified, and (2) reducing it to a computational assumption through *game transformations*.

Regarding unpredictability for a function f, the game $\mathsf{unpredictability}(f)$ is defined as follows [Now08]: a *seed* is picked at random in the set of seeds G ($\mathbb{Z}_n^*$ in the case of BBS); a sequence of bits $[b_0,\ldots,b_{len}]$ is computed by f; this sequence, deprived of its first bit b_0, is passed to the attacker A; the latter returns its guess $\widehat{b_0}$. The result of the game is whether the guess is right or not. To define unpredictability for assembly code, one needs to lift the previous definition because it applies to mathematical functions without any consideration for their implementation. This makes a difference because, contrary to mathematical functions, assembly code does not work as intended for arbitrary input, due to restrictions imposed by the choice of implementation. The basic idea is thus to extract from the assembly code its semantics in terms of a mathematical function and to inject it into the definition of unpredictability: $\mathsf{unpredictability\_assembly}(c) =_{def} \mathsf{unpredictability}([\![c]\!])$. For $[\![c]\!]$ to be well-defined, the assembly code c has to be deterministic and terminating, and, more importantly, one needs to make clear under which restrictions the assembly code behaves as intended, i.e., correctness.

The advantage of the lifting explained above is that it makes clear how to organize formal verification of assembly code with cryptographic proofs. Games for assembly connect to standard games through *implementation steps*, that can be justified formally by ensuring determinism, termination, and correctness. Since implementation steps come in addition to the other types of game transformations [Sho04], this makes it easier to develop a formal framework for verification of assembly with cryptographic proofs: pick up a formal framework for game-based proofs and a formal framework to verify assembly, and add the machinery to perform implementation steps. The rest of this paper explains how we extended [AM06] and integrated it with [Now07] for this purpose.

4 Verification of Functional Correctness of Assembly

To perform cryptographic proofs of an implementation, we need in particular to prove its functional correctness. This is technically difficult for assembly because handling of jumps results in non-standard logics, usually verbose, and thus less practical than standard Hoare logic. To overcome this difficulty, we formalize the proof-carrying code framework of [SU07] that provides not only a compositional operational semantics and Hoare logic for assembly with jumps, but also shows that derivations for this non-standard operational semantics and this Hoare logic can be obtained from standard operational semantics and standard Hoare logic by compilation.

4.1 Operational Semantics

Formalization of States A state consists of a *store* and a *heap*. The store is a finite map from registers to integers of finite size. Let int_n be the type of machine integers encoded with n bits. Most registers contain values of type int_{32} (the exception is the *extended accumulator* of type

int_n with $n \geq 8$). We have the following notations: $[\![r]\!]_s$ is the value of register r in store s; $s\{v/r\}$ is the store resulting from updating register r with value v in store s. The heap is a finite map from locations to integers of type int_{32}. The heap is tailored to word-accesses because most memory accesses in our applications are word-aligned. We have the following notation: $h[l]$ is the contents of location l of the heap h; it is `None` when the location is undefined.

We found it convenient to separate general-purpose registers from the *multiplier* (the subset of registers dedicated to arithmetic computations), hence the following definition of states: $state =_{def} store \times multiplier \times heap$. States are finally extended with a *label* (that represents the value of the program counter of the instruction being currently executed) and can be *error states* (because some instructions may trap). We distinguish error states using an `option` type, hence the definition of labelled states: $lstate =_{def} \texttt{option}\ (label \times state)$.

One-step, Non-branching Instructions The semantics of non-branching MIPS instructions is a predicate noted $s - i \longrightarrow s'$ where i is a MIPS instruction, s (resp. s') is the state before (resp. after) its execution. When formalizing the semantics of instructions, we need to express conditions such as word-alignment, absence of arithmetic overflow, etc. These conditions require manipulations such as sign-extending int_{16} integers to int_{32} integers, checking for divisibility by 4, etc. For this purpose, we introduce various operators: $(v)_{int_{16}\to int_{32}}$ sign-extends the value v from 16 to 32 bits, $(v)_{32\to\mathbb{N}}$ interprets the value v as an unsigned integer, etc.

Figure 3 illustrates the semantics of MIPS instructions with the rules for the instruction `lw` ("load word"). There are two rules depending on whether the memory access is word-aligned and the accessed location is undefined. (The notation $+_h$ is the addition of finite-size integers.)

$$\frac{\left([\![base]\!]_s +_h (off)_{int_{16}\to int_{32}}\right)_{32\to\mathbb{N}} = 4\times p \quad h[p] = \texttt{Some}\ z}{\texttt{Some}\ (s,m,h) - \texttt{lw}\ rt\ off\ base \longrightarrow \texttt{Some}\ (s\{z/rt\},m,h)}\ \text{exec0\_lw}$$

$$\frac{\forall p.\ \left([\![base]\!]_s +_h (off)_{int_{16}\to int_{32}}\right)_{32\to\mathbb{N}} \neq 4\times p \ \vee\ h[p] = \texttt{None}}{\texttt{Some}\ (s,m,h) - \texttt{lw}\ rt\ off\ base \longrightarrow \texttt{None}}\ \text{exec\_lw\_error}$$

Figure 3: Semantics of `lw`

Big-step Operational Semantics of Assembly with Jumps Following [SU07], an assembly program is formalized as a set of labelled instructions. The latter are either labelled MIPS instructions or jump instructions (unconditional jumps `jmp` l or conditional jumps `cjmp` $b\ l$). Conditional jumps comprise MIPS instructions such as `bne` ("branch if not equal"), etc; these instructions are parameterized by conditions noted b. $\mathrm{dom}(c)$ is the set of the labels of the instructions of the assembly program c. Labelled instructions are assembled using $\oplus$.

The operational semantics of assembly programs is a predicate noted $s \succ c \twoheadrightarrow s'$ where c is a set of labelled instructions, s (resp. s') is the state before (resp. after) its execution. It is defined inductively by the rules of Fig. 4. These rules are a generalization of [SU07] with more instructions and with error states. The originality of this semantics can be appreciated by looking at the two rules for sequences using $\oplus$; intuitively, they are a mix of the rules for sequence and while-loops of traditional Hoare logic.

$$\frac{\texttt{Some}\ s - i \longrightarrow \texttt{Some}\ s'}{\texttt{Some}\ (l,s) \succ l : i \twoheadrightarrow \texttt{Some}\ (l+1,s')} \qquad \frac{\texttt{Some}\ s - i \longrightarrow \texttt{None}}{\texttt{Some}\ (l,s) \succ l : i \twoheadrightarrow \texttt{None}}$$

$$\frac{l \neq l'}{\texttt{Some}\ (l,s) \succ l : \texttt{jmp}\ l' \twoheadrightarrow \texttt{Some}\ (l',s)}$$

$$\frac{[\![b]\!]_s \quad l \neq l'}{\texttt{Some}\ (l,s) \succ l : \texttt{cjmp}\ b\ l' \twoheadrightarrow \texttt{Some}\ (l',s)} \qquad \frac{\neg[\![b]\!]_s}{\texttt{Some}\ (l,s) \succ l : \texttt{cjmp}\ b\ l' \twoheadrightarrow \texttt{Some}\ (l+1,s)}$$

$$\frac{l \in \mathrm{dom}(c_1) \quad \texttt{Some}\ (l,s) \succ c_1 \twoheadrightarrow s' \quad s' \succ c_1 \oplus c_2 \twoheadrightarrow s''}{\texttt{Some}\ (l,s) \succ c_1 \oplus c_2 \twoheadrightarrow s''} \qquad \frac{l \in \mathrm{dom}(c_2) \quad \texttt{Some}\ (l,s) \succ c_2 \twoheadrightarrow s' \quad s' \succ c_1 \oplus c_2 \twoheadrightarrow s''}{\texttt{Some}\ (l,s) \succ c_1 \oplus c_2 \twoheadrightarrow s''}$$

$$\frac{}{\texttt{None} \succ c \twoheadrightarrow \texttt{None}} \qquad \frac{l \notin \mathrm{dom}(c)}{\texttt{Some}\ (l,s) \succ c \twoheadrightarrow \texttt{Some}\ (l,s)}$$

Figure 4: Big-step Operational Semantics of Assembly with Jumps

4.2 Hoare Logics

The non-standard operational semantics of the previous section gives rise to a non-standard Hoare logic for assemblies with jumps. We now formalize the Hoare logics from [SU07] (actually, extensions known as Separation Logic [Rey02]).

Assertions Properties of states are specified using a shallow-encoding of the logical connectives of Separation Logic, i.e., assertions are functions from states to the type **Prop** of propositions in Coq [AM06]: $assertion =_{def} store \to multiplier \to heap \to \mathbf{Prop}$. The satisfiability of an assertion can depend on the value of the current label: $assn =_{def} label \to assertion$.

Standard Separation Logic A triple in this logic is noted $\{\mathscr{P}\}c\{\mathscr{Q}\}$ where $\mathscr{P}$ and $\mathscr{Q}$ are assertions and c is an assembly program with while-loops instead of jumps. Let us introduce a function that computes the weakest precondition of one-step, non-branching MIPS instructions. Here is an excerpt of this function for the "load word" instruction:

$$\mathscr{W}\,\mathscr{P}\,i\,\mathscr{Q} =_{def} \textbf{match}\ i\ \textbf{with}$$
$$\mid \texttt{lw}\ rt\ off\ base \Rightarrow \lambda s.\exists p.\left([\![base]\!]_s +_h ([\![off]\!]_s)_{int_{16}\to int_{32}}\right)_{32\to\mathbb{N}} = 4\times p \wedge \exists z.h[p] = \texttt{Some}\ z \wedge \mathscr{Q}\,s\{z/rt\}$$
$$\mid \ldots\ \textbf{end}$$

Using the above function, the standard separation logic is defined by the following rules:

$$\frac{}{\{\mathscr{W}\,\mathscr{P}\,i\,Q\}\,i\,\{\mathscr{Q}\}} \qquad \frac{\{\mathscr{P}\}c_1\{\mathscr{R}\} \quad \{\mathscr{R}\}c_2\{\mathscr{Q}\}}{\{\mathscr{P}\}c_1\,;c_2\{\mathscr{Q}\}} \qquad \frac{\{\lambda s.\mathscr{P}\wedge[\![b]\!]_s\}c\{\mathscr{P}\}}{\{\mathscr{P}\}\,\texttt{while}\,b\,c\{\lambda s.\mathscr{P}\wedge\neg[\![b]\!]_s\}}$$

$$\frac{\{\lambda s.\mathscr{P}\wedge[\![b]\!]_s\}c_1\{\mathscr{Q}\} \quad \{\lambda s.\mathscr{P}\wedge[\![b]\!]_s\}c_2\{\mathscr{Q}\}}{\{\mathscr{P}\}\,\texttt{if}\,b\,\texttt{then}\,c_1\,\texttt{else}\,c_2\{\mathscr{Q}\}} \qquad \frac{\mathscr{P}\to\mathscr{P}' \quad \{\mathscr{P}'\}c\{\mathscr{Q}'\} \quad \mathscr{Q}'\to\mathscr{Q}}{\{\mathscr{P}\}c\{\mathscr{Q}\}}$$

This logic is formally proved sound and complete w.r.t. standard big-step operational semantics, where assembly code with while-loops is built out of MIPS instructions (Sect. 4.1), sequences ($c_1 ; c_2$), structured branching ($\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2$), and while-loops ($\texttt{while}\ b\ c$):

$$\frac{}{\texttt{None} - c \twoheadrightarrow \texttt{None}} \qquad \frac{s - c \longrightarrow s'}{s - c \twoheadrightarrow s'} \qquad \frac{s - c_1 \twoheadrightarrow s'' \quad s'' - c_2 \twoheadrightarrow s'}{s - c_1 ; c_2 \twoheadrightarrow s'}$$

$$\frac{[\![b]\!]_s \qquad \texttt{Some}\ (s,m,h) - c_1 \twoheadrightarrow s'}{\texttt{Some}\ (s,m,h) - \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \twoheadrightarrow s'} \qquad \frac{\neg[\![b]\!]_s \qquad \texttt{Some}\ (s,m,h) - c_2 \twoheadrightarrow s'}{\texttt{Some}\ (s,m,h) - \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \twoheadrightarrow s'}$$

$$\frac{[\![b]\!]_s \quad \texttt{Some}\ (s,m,h) - c \twoheadrightarrow s' \quad s' - \texttt{while}\ b\ c \twoheadrightarrow s''}{\texttt{Some}\ (s,m,h) - \texttt{while}\ b\ c \twoheadrightarrow s''} \qquad \frac{\neg[\![b]\!]_s}{\texttt{Some}\ (s,m,h) - \texttt{while}\ b\ c \twoheadrightarrow \texttt{Some}\ (s,m,h)}$$

Separation Logic based on the compositional Hoare logic of [SU07] A triple in this logic is noted $[\mathscr{P}]\,c\,[\mathscr{Q}]$ where $\mathscr{P}$ and $\mathscr{Q}$ are labelled assertions (type *assn*) and c is an assembly program with jumps. We introduce predicate transformers that enforce assertions to be satisfiable for labels inside (resp. outside) a domain: $\mathscr{P}|_d =_{def} \lambda l.\mathscr{P}\ l \wedge l \in d$, $\mathscr{P}|_{\overline{d}} =_{def} \lambda l.\mathscr{P}\ l \wedge l \notin d$. Using above predicate transformers and the above weakest-precondition function, we formalize the rules for the compositional Hoare logic below. This logic is formally proved sound and complete w.r.t. the big-step operational semantics of Sect. 4.1.

$$\frac{}{\left[\begin{array}{ll}\lambda pc.\lambda s. & pc = l \wedge (\mathscr{P}\ j\ s \vee j = l) \vee \\ & pc \neq l \wedge \mathscr{P}\ pc\ s\end{array}\right] l : \texttt{jmp}\ j \left[\mathscr{P}\right]}$$

$$\frac{}{\left[\begin{array}{ll}\lambda pc.\lambda s. & pc = l \wedge (\neg[\![b]\!]_s \wedge \mathscr{P}\ (l+1)\ s \vee [\![b]\!]_s \wedge (\mathscr{P}\ j\ s \vee j = l)) \vee \\ & pc \neq l \wedge \mathscr{P}\ pc\ s\end{array}\right] l : \texttt{cjmp}\ b\ j \left[\mathscr{P}\right]}$$

$$\frac{}{[\mathscr{P}]\,nop\,[\mathscr{P}]} \qquad \frac{}{\left[\begin{array}{ll}\lambda pc.\lambda s. & pc = l \wedge \mathscr{W}\ \mathscr{P}\ c\ (\mathscr{P}\ (l+1))\ s \vee \\ & pc \neq l \wedge \mathscr{P}\ pc\ s\end{array}\right] l : c \left[\mathscr{P}\right]}$$

$$\frac{\left[\mathscr{P}|_{\mathrm{dom}(c_1)}\right] c_1 \left[\mathscr{P}\right] \qquad \left[\mathscr{P}|_{\mathrm{dom}(c_2)}\right] c_2 \left[\mathscr{P}\right]}{\left[\mathscr{P}\right] c_1 \oplus c_2 \left[\mathscr{P}|_{\overline{\mathrm{dom}(c_1 \oplus c_2)}}\right]} \qquad \frac{\forall l.\mathscr{P}\ l \rightarrow \mathscr{P}'\ l \qquad \left[\mathscr{P}'\right] c \left[\mathscr{Q}'\right] \qquad \forall l.\mathscr{Q}'\ l \rightarrow \mathscr{Q}\ l}{\left[\mathscr{P}\right] c \left[\mathscr{Q}\right]}$$

4.3 Compilation from Standard Semantics and Hoare Logic

[SU07] shows that derivations for the previous non-standard operational semantics and Hoare logic can also be obtained from standard operational semantics and Hoare logic through compilation. This is a result of interest because it allows us to work with standard operational semantics and Hoare logic (that are more practical to deal with formally) while still being able to recover formal proofs for assembly with jumps (these are the formal proofs that we really want, for example for shipping in a proof-carrying code scenario).

The compilation procedure turns if-then-else's and while-loops into conditional and unconditional jumps. The compilation of program c with while-loops to an assembly program c' with jumps is noted $c\ {}^{l}\!\searrow_{l'}\ c'$ where l (resp. l') is the start (resp. end) label of the compiled program:

$$\frac{}{i \;{}^{l}\!\searrow_{l+1}\; l : i} \qquad \frac{c_1 \;{}^{l_1+1}\!\searrow_{l_2}\; c_1' \quad c_2 \;{}^{l+1}\!\searrow_{l_1}\; c_2'}{\texttt{if}\; b \;\texttt{then}\; c_1 \;\texttt{else}\; c_2 \;{}^{l}\!\searrow_{l_2}\; l : \texttt{cjmp}\; b\; (l_1+1) \oplus ((c_2' \oplus l_1 : \texttt{jmp}\; l_2) \oplus c_1')}$$

$$\frac{c_1 \;{}^{l}\!\searrow_{l_1}\; c_1' \quad c_2 \;{}^{l_1}\!\searrow_{l_2}\; c_2'}{c_1 ; c_2 \;{}^{l}\!\searrow_{l_2}\; c_1' \oplus c_2'} \qquad \frac{c \;{}^{l+1}\!\searrow_{l_1}\; c'}{\texttt{while}\; b\; c \;{}^{l}\!\searrow_{l_1+1}\; l : \texttt{cjmp}\; (\neg b)\; (l_1+1) \oplus (c' \oplus l_1 : \texttt{jmp}\; l)}$$

Through compilation, derivations of operational semantics can be compiled from the standard one to the non-standard one of Sect. 4.1, and, similarly, proofs in Separation Logic can be compiled from the standard one to the non-standard one of Sect. 4.2:

Lemma *preservation_of_evaluations* :
for all $c\ s\ l\ c'\ s'\ l'$, if $c \;{}^{l}\!\searrow_{l'}\; c'$ and $\texttt{Some}\ s - c \twoheadrightarrow \texttt{Some}\ s'$, then
$\texttt{Some}\ (l,s) \succ c' \twoheadrightarrow \texttt{Some}\ (l + \mathrm{card}(\mathrm{dom}(c')), s')$.
Lemma *preservation_hoare* :
for all $\mathscr{P}, \mathscr{Q}, c$ such that $\{\mathscr{P}\}\, c\, \{\mathscr{Q}\}$ and for all l, c', l' such that $c \;{}^{l}\!\searrow_{l'}\; c'$ then
$[\lambda pc.\lambda s.\ pc = l \wedge \mathscr{P}\ s]\, c'\, [\lambda pc.\lambda s.\ pc = l' \wedge \mathscr{Q}\ s]$.

5 Extraction of the Semantics of BBS in Assembly

5.1 The Functional Correctness of BBS in Assembly

Let us provide two functions encode and decode such that: $\mathsf{encode}(n,k,seed,m)$ builds a state from the requested number n of pseudoramdom 32-bits words, the number k of 32-bits words reserved for the encoding of the seed and the modulus, the *seed*, and the modulus m; and $\mathsf{decode}(s)$ is the list of pseudorandom bits stored in the state s. These functions impose a specific memory layout depicted in Fig. 5. Besides the encoding of the seed (in memory area X) and the modu-

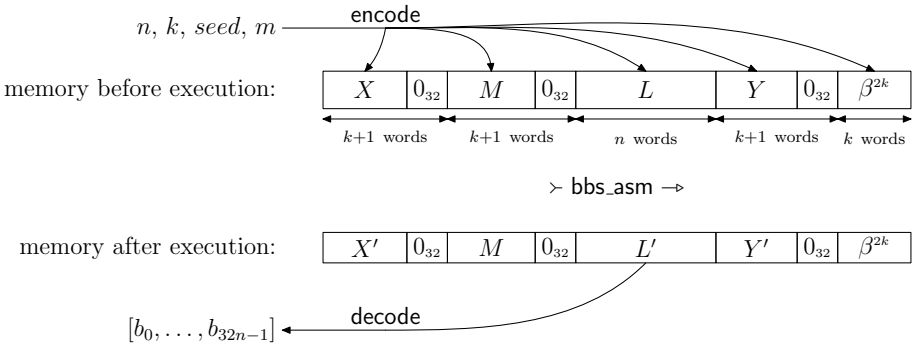


Figure 5: Encoding and decoding of input/output

lus (in M) as multi-precision integers, and initialization of the list of pseudorandom bits (in L) to an appropriate length, encode provides additional storage (Y, β^{2k}, trailing words initialized to 0_{32}) specific to our implementation (this stems from our Montgomery multiplication). Note

that, as long as $4(4k+n+2) < 2^{32}$, n and k can be very large, k effectively covering lengths for which the quadratic residuosity problem is indeed believed to be intractable. This is one desirable side-effect of our approach to precisely pinpoint the range of k.

Using above functions, the verification goal is stated below:

$$4(4k+n+2) < 2^{32} \rightarrow$$
$$\left[\begin{array}{c}\lambda pc.\lambda s.\ pc = 0\ \wedge \\ \mathsf{encode}(n,k,seed,m) = s\end{array}\right] \mathsf{bbs\_asm} \left[\begin{array}{c}\lambda pc.\lambda s.\ pc = 240\ \wedge \\ \mathsf{decode}(s) = \mathsf{bbs\_fun}(32\times n, seed, m)\end{array}\right]$$

Starting from an appropriate encoding of the inputs, the execution of bbs_asm leads to a final state from which one can extract the intended list of pseudorandom bits (we display a simplified statement; the complete statement can be found in [Code]). It is here that the restrictions imposed by the choice of implementation mentioned in Sect. 3 appear, for the above triple cannot be proved for arbitrary values of n and k.

In practice, we conduct formal proof using standard Separation Logic and obtain the triple above by applying the lemma *preservation_hoare* of Sect. 4.3. The effort therefore concentrates on the following triple where the assembly program with jumps has been replaced by its (manually) "decompiled" version (with if-then-else's and while-loops):

$$4(4k+n+2) < 2^{32} \rightarrow$$
$$\{\lambda s.\ \mathsf{encode}(n,k,seed,m) = s\}\,\mathsf{bbs\_asm\_decompile}\,\{\lambda s.\ \mathsf{decode}(s) = \mathsf{bbs\_fun}(32\times n, seed, m)\} \quad (1)$$

Note that we are dealing with a generalized version of the BBS algorithm (bbs_fun takes the modulus m in $\mathbb{Z}$, whereas bbs in Sect. 2.1 uses the types $\mathbb{Z}_m^*$ and QR_m):

$$\mathsf{bbs\_fun}(len \in \mathbb{N}, seed \in \mathbb{Z}, m \in \mathbb{Z}) =_{def} \mathsf{bbs\_fun\_rec}(len, seed^2 \pmod m, m)$$
$$\mathsf{bbs\_fun\_rec}(len \in \mathbb{N}, x \in \mathbb{Z}, m \in \mathbb{Z}) =_{def}$$
$$\textbf{match } len \textbf{ with } 0 \Rightarrow [] \mid len'+1 \Rightarrow \mathsf{parity}(x) :: \mathsf{bbs\_fun\_rec}(len', x^2 \pmod m, m) \textbf{ end}$$

This is a sound generalization because the information that $\mathbb{Z}_m^*$ is a cyclic group is not needed in the proof of functional correctness (only in the cryptographic proof).

5.2 Extraction of the Semantics of BBS in Assembly

First, we prove that bbs_asm is terminating and deterministic, i.e., for all n, k, *seed* and m, there exists a unique state s' such that $\texttt{Some}\ (0, \mathsf{encode}(n,k,seed,m)) \succ \mathsf{bbs\_asm} \twoheadrightarrow s'$. From the Separation Logic triple of the previous section, we derive by soundness of the Separation Logic:

Lemma *correctness* :
if $\texttt{Some}\ (0, \mathsf{encode}(n,k,seed,m)) \succ \mathsf{bbs\_asm} \twoheadrightarrow \texttt{Some}\ (l', s')$ and $4(4k+n+2) < 2^{32}$,
then $l' = 240$ and $\mathsf{decode}(s') = \mathsf{bbs\_fun}(32\times n, seed, m)$

Then comes the proof of termination. First, we prove that there is a final state, without proving whether it is an error state or not:

Lemma *execution_bbs_asm* :
if $4(4k+n+2) < 2^{32}$, then there exists s' s. t. $\texttt{Some}\ (0, \mathsf{encode}(n,k,seed,m)) \succ \mathsf{bbs\_asm} \twoheadrightarrow s'$

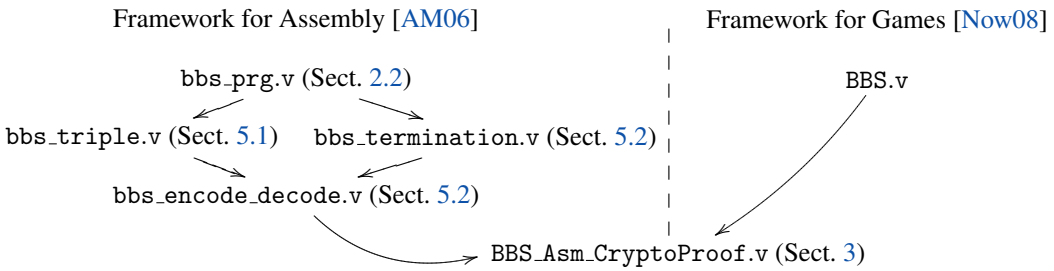
This is proved by induction on the variant of the outermost loop, and then on nested loops. Second, by the triple (1), we derive the fact that this final state cannot be an error state.

The lemmas above allow us to define a function $\mathsf{exec}_{\mathsf{bbs\_asm}}$ that maps a number n of 32 bits words, a number k of 32 bits words, a *seed* and a modulus m to a state from which one can extract the desired pseudorandom bits, so that the semantics of bbs_asm can be written as a mathematical function:

$$[\![\mathsf{bbs\_asm}]\!] =_{def} \mathsf{prefix}_{len+1}\left(\mathsf{decode}\left(\mathsf{exec}_{\mathsf{bbs\_asm}}\left(\left\lceil\frac{len+1}{32}\right\rceil, \lceil\log_{2^{32}}(m)\rceil, seed, m\right)\right)\right)$$

Since bbs_asm always return a number of pseudorandom bits that is a multiple of 32, we need to take a prefix of its output; $len+1$ is the length requested by the unpredictability game.

Putting it All Together The Figure below summarizes how we organize the complete game-based proof of BBS in assembly. `bbs_encode_decode.v` contains the encode/decode functions and the formal proof of correctness and termination, derived from the formal proof of the Separation Logic triple for bbs_asm. `BBS_Asm_CryptoProof.v` contains the game-based proof, making use of the correctness and termination lemmas and of the cryptographic proof of the BBS algorithm provided by `BBS.v`, taken directly from [Now08].



6 Technical Aspects of the Coq Formalization

The formalization of assembly programs, operational semantics, Separation Logic, as well as all supporting lemmas is the result of a revision of our previous work [AM06]. This revision was made necessary to address scalability issues. We do not comment extensively about this revision except to say that we used SSReflect [GM07], a recently publicized Coq extension, that favors a proof style that naturally led to shorter proof scripts (for illustration, proof scripts of experiments in [AM06] shrank by 70% in terms of lines of code).

The new aspect of our framework is the formalization of the proof-carrying code framework of [SU07], that we instantiate to Separation Logic and MIPS instructions and extend to deal with error-states. Table 1 makes it clear what is formalized w.r.t. [SU07]. In brief, what we do not do: we do not formalize Section 5 of [SU07] and we formalize only the so-called "non-constructive proofs" of Theorems 17 and 18 (indeed, for these two theorems, the proofs come in two flavors).

The formal proof of the Separation Logic triple of Sect. 5.1 is technically the most demanding part of the proof effort. Our assembly program of BBS is large (at least by the current standards

Reference in [SU07]	Reference in [Code] and status	Proof script size
Section 2	file `goto.v`	460 lines
Figure 1, Lemma 1, 3	Done	
Lemma 2	Particular cases only	
Section 3	file `sgoto.v`	745 lines
Section 3.1: Figure 2, Lemmas 4–5, Theorems 6–8, Corollary 9	Done	
Section 3.2: Figure 3, Theorem 10, Lemma 11, Theorem 12	Done	
Section 4	file `compile.v`	1429 lines
Section 4.1: Figure 5, Lemmas 13–14, Theorems 15–16	Done	
Section 4.2: Theorems 17–18	Done	
Section 4.3	Done, file `sgoto_hoare.v`	371 lines
Section 5	Not done	
Appendix A	Done (revision of [AM06])	
	file `state.v`	615 lines
	file `mips.v`	881 lines
	file `mips_hoare.v`	1045 lines
Appendix B		
Theorems 6–7, 15–18	Done (spread over above files)	

Table 1: Formalization of [SU07]

of proof assistant-based verification [AM06, MG07]): 239 instructions that spread over several snippets of code. Table 2 makes it precise which snippets are used and their respective size.

Function	Reference in [Code]	Program size
BBS	`bbs_prg.v`	14 insns
Montgomery strict (Fig. 2)	`mont_mul_strict_prg.v`	9 insns
Montgomery raw ([AM06])	`mont_mul_prg.v`	36 insns
Multi-precision subtraction	`multi_sub_prg.v`	20 insns
Multi-precision comparison	`multi_lt_prg.v`	11 insns
Array initialization	`multi_zero_prg.v`	6 insns

Table 2: The assembly code of BBS in Coq

Table 3 summarizes the size of proof scripts used in the proof of the Separation Logic triple of BBS. It is always difficult to comment about the size of proof scripts because we are lacking good metrics for comparison. Yet, looking at related work, it is fair to claim that our framework for formal proof of assembly programs allows for short proof scripts: this can be appreciated by looking at several similar experiments in common among the work in this paper and [AM06, MG07] (verification of multi-precision arithmetic, Montgomery multiplication, but also Montgomery exponentiation, not used in this paper though).

Function	Reference in [Code]	Size
BBS	`bbs_triple.v`	841 lines
Montgomery strict	`mont_{mul,square}_strict_init_triple.v`	601 lines
Montgomery raw	`mont_{mul,square}_triple.v`	1205 lines
Multi-precision subtraction	`multi_sub_inplace_left_triple.v`	506 lines
Multi-precision comparison	`multi_lt_triple.v`	405 lines
Array initialization	`multi_zero_triple.v`	129 lines
Total		3687 lines

Table 3: Formal proof of the Separation Logic triple of BBS

7 Conclusion

We addressed the problem of certification of assembly code with cryptographic proofs. We proposed an approach that extends game-based proofs so as to integrate formal proofs of functional correctness with formal cryptographic proofs in a clear way, understandable by both cryptographers and implementers. Our proposition is supported by a concrete framework developed in the Coq proof assistant. As an illustration, we provided the first assembly program for a pseudorandom number generator that is certified with a cryptographic proof.

Future Work The cryptographic proof of BBS on which we rely in this paper is asymptotic. It shows that the probability that an attacker predicts the next bit can be made arbitrarily small, but it does not give any concrete value for the security parameter. One possible extension of our approach would be to link our assembly implementation of BBS to a cryptographic proof of the *concrete security* of BBS.

Our certified implementation of BBS could be used as the source of pseudorandomness in the implementation of further cryptographic primitives. Indeed, even though it is probabilistic, such a primitive is still deterministic in the sense that for any two equal inputs it outputs the same distribution; one can thus extract its semantics as a mathematical function and inject it into the appropriate standard definition of security (such as *semantic security* in the case of ElGamal).

Acknowledgements: This work was partially supported by KAKENHI 21700048 and 21500046.

Bibliography

[ATM07] Affeldt, R., Tanaka, M., Marti, N.: Formal Proof of Provable Security by Game-Playing in a Proof Assistant. Int. Conf. on Provable Security. LNCS, vol. 4784, pp. 151–168. Springer (2007).

[AM06] Affeldt, R., Marti, N.: An Approach to Formal Verification of Arithmetic Functions in Assembly. Annual Asian Computing Science Conference, Dec. 2006. LNCS, vol. 4435, pp. 346–360. Springer, Heidelberg (2008).

[BBU08] Backes, M., Berg, M., Unruh D.: A Formal Language for Cryptographic Pseudocode. Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning. LNCS, vol. 5330, , pp. 353–376. Springer (2008).

[BGZ09] Barthe, G., Grégoire, B., Zanella, S.B.: Formal certification of code-based cryptographic proofs. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp.90–101. ACM Press.

[BR04] Bellare, M., Rogaway, P.: Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004.

[BBS86] Blum, L., Blum, M., Shub, M.: A simple unpredictable pseudo random number generator. SIAM Journal on Computing, 15(2):364–383. Society for Industrial and Applied Mathematics, 1986.

[Code] Affeldt, R., Nowak D., Yamada K.: Certifying Assembly with Cryptographic Proofs: the Case of BBS. http://staff.aist.go.jp/reynald.affeldt/bbs

[GM07] Gonthier, G., Mahboubi, A.: A Small Scale Reflection Extension for the Coq System. Technical Report 6455, Dec. 2007. INRIA.

[Mips] MIPS Technologies. MIPS32 4KS Processor Core Family Software User's Manual MIPS Technologies, Inc., 1225 Charleston Road, Mountain View, CA 94043-1353.

[Mon85] Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation, 44(170):519–521, 1985.

[MG07] Myreen, M.O., Gordon, M.J.C.: Verification of Machine Code Implementations of Arithmetic Functions for Cryptography. Theorem Proving in Higher Order Logics: Emerging Trends Proceedings. Internal Report 364/07, Aug. 2007. Department of Computer Science, University of Kaiserslautern.

[MSG09] Myreen, M.O., Slind, K., Gordon, M.J.C.: Extensible proof-producing compilation. Int. Conf. on Compiler Construction. LNCS, vol. 5501, pp. 2–16. Springer (2009).

[Now07] Nowak, D.: A framework for game-based security proofs. Int. Conf. on Information and Communications Security. LNCS, vol. 4861, pp. 319–333. Springer (2007).

[Now08] Nowak, D.: On formal verification of arithmetic-based cryptographic primitives. Int. Conf. on Information Security and Cryptology, Dec. 2008. LNCS, vol. 5461, pp. 368-382. Springer (2009).

[Rey02] Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. IEEE Symp. on Logic in Computer Science, pp. 55–74 (2002). Invited lecture.

[SU07] Saabas, A., Uustalu, T.: A compositional natural semantics and Hoare logic for low-level languages. Theoretical Computer Science 373(3), 273–302. Elsevier (2007).

[Sho04] Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.

[Yao82] Yao, A.C.: Theory and applications of trapdoor functions. IEEE Annual Symp. on Foundations of Computer Science. pp. 80–91. IEEE (1982).

An Extension of the Inverse Method to Probabilistic Timed Automata

Étienne André[1], Laurent Fribourg[1] and Jeremy Sproston[2]

[1] LSV – ENS de Cachan & CNRS, France
[2] Dipartimento di Informatica, Università di Torino, Italy

Abstract: Probabilistic timed automata can be used to model systems in which probabilistic and timing behavior coexist. Verification of probabilistic timed automata models is generally performed with regard to a single reference valuation of the timing parameters. Given such a parameter valuation, we present a method for obtaining automatically a constraint on timing parameters for which the reachability probabilities (1) remain invariant and (2) are equal to the reachability probabilities for the reference valuation. The method relies on parametric analysis of a non-probabilistic version of the probabilistic timed automata model using the "inverse method". Our approach is useful for avoiding repeated executions of probabilistic model checking analyses for the same model with different parameter valuations. We provide examples of the application of our technique to models of randomized protocols.

Keywords: Probabilistic Model Checking, Parametric Timed Automata, Randomized Protocols

1 Introduction

Timed automata are finite control automata equipped with *clocks*, which are real-valued variables which increase uniformly [AD94]. This model is useful for reasoning about real-time systems, because one can specify quantitatively the interval of time during which the transitions can occur, using the bounds involved in invariants and guards labeling the nodes and arcs of the automaton. An extension of timed automata to the probabilistic framework, where discrete actions are replaced by discrete probability *distributions* over discrete actions, has been defined in [Jen96, KNSS02]. This model has been applied to a number of case studies [KNPS06]. Model-checking analysis of probabilistic timed automata normally proceeds by reducing the model to a finite-state probabilistic system and then employing a probabilistic model-checking tool such as PRISM [HKNP06, wp].

The constants used in some timing constraints of a real-time system may not be known, or may be known with some uncertainty. Therefore methods for automatically generating values on parameters in timing constraints for which the system behaves correctly are desirable. Methods for synthesizing such parameters in timed automata have first been presented in [AHV93]. In [ACEF], the following *inverse problem* has been considered: given a parametric timed automaton and a *reference valuation*, which is a particular valuation of the parameters of the model, find a constraint K_0 on the parameters which is satisfied by the reference valuation and in which the

model behaves in the same manner as in the case of the reference valuation. For example, if the reference valuation is known to exhibit good behavior, such as the impossibility of reaching an error state, then we aim to find a constraint on the parameters within which such good behavior is guaranteed. In particular, this allows the system designer to optimize some parameters of the system.

In this paper, we consider the application of the inverse method to *probabilistic* timed automata models. We aim at synthesizing a constraint such that, for any valuation of the parameters satisfying K_0, the model is "time-abstract" equivalent to the model for the reference valuation. In the context of probabilistic timed automata, the computed constraint K_0 defines parameter valuations for which, in particular, minimum (resp., maximum) probabilities of satisfying a given property (e.g., reachability of a certain location) are all equal. Therefore, given the computation of K_0, it suffices to compute a minimum (resp., maximum) probability for a single parameter valuation satisfying K_0. In order to infer such a constraint K_0, we transform the system into a *non-probabilistic* timed automaton, and apply the original inverse method of [ACEF] to this timed automaton.

Motivation. This approach is particularly important for probabilistic timed automata for the following reason. As mentioned above, model checking for probabilistic timed automata in practice generally relies on the reduction of the model to a finite-state system. The effectiveness of the discrete-time semantics method most commonly used is sensitive to the timing constants used in the description of the model: more precisely, the greater the timing constants, the larger the state space of the finite-state system obtained from the discrete-time semantics construction, and the more difficult the verification (the zone-based algorithm of [KNSW07] does not have this property, but does not always perform better in practice than the discrete-time semantics approach). In case studies it is standard to rescale the time unit used in the model to reduce the magnitude of the timing constraints in order to reduce the size of the resulting finite-state system. This rescaling operation possibly involves rounding lower bounds on clocks downwards, and upper bounds upwards [KNPS06], which results in an abstraction in which the computed maximum (minimum, respectively) probabilities may be greater (or less than, respectively) the actual probabilities in the original model. The inverse method presents an alternative to this rescaling approach. By applying the inverse method to obtain the constraint K_0, we can choose the parameter valuation satisfying K_0 with the lowest possible values for the timing constraints: then, by performing analysis on the model with this parameter valuation, we can obtain the same minimum and maximum probabilities as on the model using the reference valuation of the parameters. Hence, as in the rescaling approach, we can obtain discrete-time models of limited size by reducing the magnitude of the timing constants; however, in contrast to the rescaling approach, we can avoid rounding of lower and upper bounds, thereby resulting in a model exhibiting the same probabilities as obtained for the model corresponding to the reference valuation. We note that this motivation also applies when considering discrete-time approaches to timed automata verification, such as in [HMP92, BMT99, Bey01].

Related Work. In contrast to [LMT03, Daw04, HKM08], we do not consider parameters over probabilities, but only over timing constraints. We consider a probabilistic parametric timed

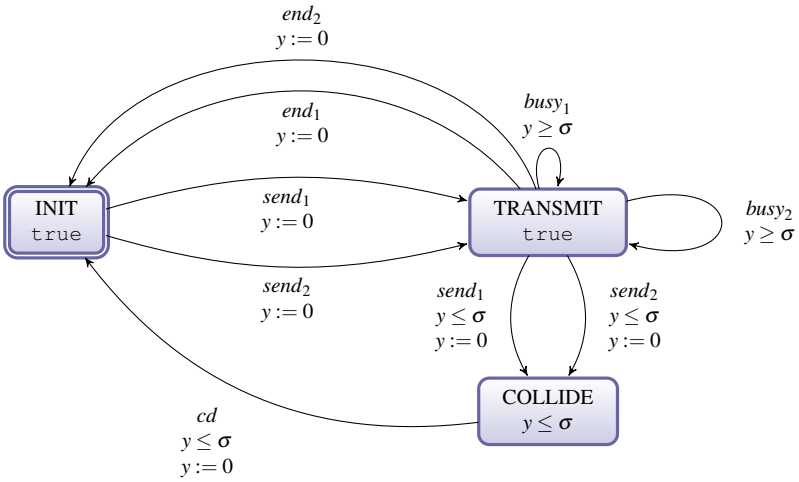


Figure 1: CSMA/CD Medium

automata framework which features both nondeterministic and probabilistic choice, rather than just probabilistic choice, as was considered in [CDF+08].

An Illustrative Example. Consider the CSMA/CD protocol, as studied in the context of probabilistic timed automata in [KNSW07]. We consider the case when there are two stations 1 and 2 trying to send data at the same time. The overall model is given by the parallel composition of three probabilistic timed automata representing the medium and two stations trying to send data. The probabilistic timed automaton representing the medium is given in Fig. 1, while the probabilistic timed automaton representing a sender i is given in Fig. 2.

The medium is initially ready to accept data from any station (event *send*). Once a station starts sending its data there is an interval of time (at most σ), representing the time it takes for a signal to propagate between the stations. In this interval the medium can accept data from the other station (resulting in a collision). After this interval, if the other station tries to send data it will get the busy signal (*busy*). When a collision occurs, there is a delay (again at most σ) before the stations realize there has been a collision, after which the medium will become free (represented by the event *cd*). If the stations do not collide, then when a station finishes sending its data (event *end*) the medium becomes idle. We model the case when initially the stations collide. A station starts by sending its data. If there is no collision, then, after λ time units, the station finishes sending its data (event *end*). On the other hand, if there is a collision (event *cd*), the station attempts to retransmit the packet, where the scheduling of the retransmission is determined by a truncated binary exponential backoff process. The delay before retransmitting is an integer number of time slots (each of length *slot*). The number of slots that the station waits after the nth transmission failure is chosen as a uniformly distributed *random* integer in the range: $0, 1, 2, \ldots, 2^{bc+1} - 1$, where $bc = \min(n, bcmax)$. This random choice is depicted in Fig. 2 by the assignment $backoff := \mathbf{RAND}(bc)$. Once this time has elapsed, if the medium appears free the station resends the data (event *send*), while if the medium is sensed busy (event *busy*)

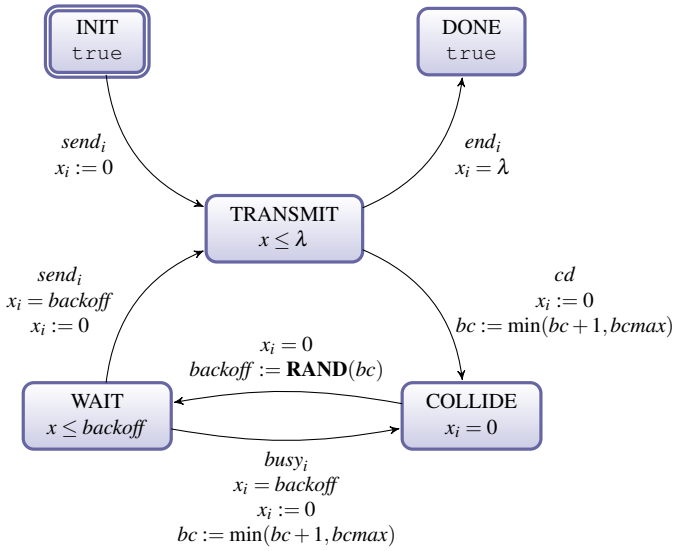


Figure 2: CSMA/CD Sender i

the station repeats this process.

The following parameters are taken from the IEEE standard 802.3 for 10 Mbps Ethernet. It constitutes the reference valuation for the following three parameters: $\sigma = 26\mu s$, $\lambda = 808\mu s$, and $slot = 2\sigma = 52\mu s$. The method for inferring a constraint K_0 on the parameters, which is satisfied by the reference valuation and in which the behavior of the model remains the same, consists in transforming the system into a non-probabilistic parametric timed automaton. We replace the random choice $backoff := \mathbf{RAND}(bc)$ with a *non-deterministic* choice, i.e., a set of 2^{bc+1} transitions associated with assignments of the form $backoff := i$, for $i = 0, 1, 2, \ldots, 2^{bc+1} - 1$. In the case where $bcmax = 1$, the application of the inverse method to the non-probabilistic parametric timed automaton (see Section 4.1) infers for K_0 the following constraint: $(0 < \sigma < slot) \wedge (15slot < \lambda < 16slot)$. In particular, the minimum and maximum probabilities for a message sent by a station to be transmitted (i.e., to reach the location DONE) after having collided exactly k times with another message (action cd) are the same under the reference valuation and another parameter valuation satisfying K_0. This has two practical implications. Firstly, in order to compute the aforementioned minimum and maximum probabilities for $\sigma = 26, \lambda = 808, slot = 52$, it suffices to compute the minimum and maximum probabilities for $\sigma = 1, \lambda = 31, slot = 2$ (because both valuations satisfy the constraint $(0 < \sigma < slot) \wedge (15slot < \lambda < 16slot)$ generated by the inverse method). Note that the valuation $\sigma = 26, \lambda = 808, slot = 52$ results in model with 5240 states, whereas the valuation $\sigma = 1, \lambda = 31, slot = 2$ results in model with 282 states. The second practical implication concerns the case in which the system designer wishes to understand the behavior of the system, in terms of minimum and maximum probabilities, for a number of parameter valuations. The approach of obtaining such information by changing manually the timing parameters and repeating model-checking analysis is potentially time consuming. Instead, the application of the inverse method shows that the minimum and maximum probabilities remain invariant for all parameter

valuations satisfying the constraint K_0.

Plan of the paper. In Section 2, we present the definition of probabilistic parametric timed automata. In Section 3, we apply the inverse method to probabilistic timed automata in the following way: we construct a non-probabilistic version of the model, which is then subject to the inverse method for parametric timed automata. As in [KNS02], some attention has to be dedicated to the non-probabilistic model construction so that the results of the inverse method apply to the original probabilistic timed automaton. In Section 4, we apply the method to three probabilistic protocols with timing parameters (CSMA/CD, IEEE 1394 root contention, IEEE 802.11 WLAN). We conclude in Section 5.

2 Parametric Probabilistic Timed Automata

In Section 2.1, we review the definition of *timed probabilistic systems*, as defined in [KNSW07], which is a variant of Segala's probabilistic timed automata [Seg95]. In Section 2.2, we extend the definition of probabilistic timed automata [KNSW07] to the parametric case, and give its semantics in terms of timed probabilistic systems (Section 2.3).

2.1 Timed Probabilistic Systems

Let AP denote a fixed finite set of atomic propositions. Let $\mathbb{R}_{\geq 0}$ be the set of non-negative real numbers. A (discrete) probability *distribution* over a countable set Z is a function $\mu : Z \to [0,1]$ such that $\sum_{z \in Z} \mu(z) = 1$. We define $\mathsf{support}(\mu) = \{z \in Z \mid \mu(z) > 0\}$. Then for an uncountable set Z we define $\mathsf{Dist}(Z)$ to be the set of functions $\mu : Z \to [0,1]$, such that $\mathsf{support}(\mu)$ is a countable set and μ restricted to $\mathsf{support}(\mu)$ is a (discrete) probability distribution. A *point distribution* is a distribution $\mu \in \mathsf{Dist}(Z)$ such that $\mu(z) = 1$ for some (unique) $z \in Z$. Often we write μ_z for the point distribution such that $\mu(z) = 1$.

A *timed probabilistic system (TPS)* is a tuple $\mathsf{T} = (S, S_0, L, Act, \Rightarrow)$ where: S is a set of *states*, including a set S_0 of *initial states*; $L : S \to 2^{AP}$ is a *state labeling* function; Act is a finite set of *actions* (disjoint from $\mathbb{R}_{\geq 0}$); $\Rightarrow \subseteq S \times \mathbb{R}_{\geq 0} \times Act \times \mathsf{Dist}(S)$ is a *probabilistic transition relation*.

We assume that the probabilistic transition relation is *total*; that is, for every state $s \in S$, there exists $(s, d, a, \mu) \in \Rightarrow$ for some $d \in \mathbb{R}_{\geq 0}, a \in Act, \mu \in \mathsf{Dist}(S)$. In Section 2.3, we give a sufficient condition on the timed probabilistic system that guarantee that such an assumption holds.

A transition $s \xrightarrow{d,a,\mu} s'$ is made from a state $s \in S$ by first nondeterministically selecting a duration-action-distribution triple (d, a, μ) such that $(s, d, a, \mu) \in \Rightarrow$, and second by making a probabilistic choice of target state s' according to distribution μ, such that $\mu(s') > 0$. An *infinite path* of a TPS is an infinite sequence of transitions $\omega = s_0 \xrightarrow{d_0,a_0,\mu_0} s_1 \xrightarrow{d_1,a_1,\mu_1} s_2 \xrightarrow{d_2,a_2,\mu_2} \cdots$. Similarly, a *finite path* of a TPS is a non-empty finite sequence of transitions $\omega = s_0 \xrightarrow{d_0,a_0,\mu_0} s_1 \xrightarrow{d_1,a_1,\mu_1} \cdots \xrightarrow{d_{n-1},a_{n-1},\mu_{n-1}} s_n$. Given a finite path $\omega = s_0 \xrightarrow{d_0,a_0,\mu_0} s_1 \xrightarrow{d_1,a_1,\mu_1} \cdots \xrightarrow{d_{n-1},a_{n-1},\mu_{n-1}} s_n$, we let $last(\omega) = s_n$. The set of finite (infinite, respectively) paths of a TPS T is denoted by $Path^{\mathsf{T}}_{fin}$ ($Path^{\mathsf{T}}_{ful}$, respectively). When clear from the context we omit the superscript T and write $Path_{fin}$ and $Path_{ful}$. We let $Path_{fin}(s)$ ($Path_{ful}(s)$, respectively) denote the set of finite (infinite,

respectively) paths commencing in the state $s \in S$.

A *scheduler* is a function which chooses an outgoing distribution in the last state of a path. Formally, a scheduler of a TPS is a function $\mathscr{S}$ mapping every finite path ω of the TPS to a triple (d,a,μ) such that $(last(\omega),d,a,\mu) \in \Rightarrow$. A scheduler resolves the nondeterminism by choosing a probability distribution based on the finite path executed so far. If a TPS is guided by scheduler σ and has the finite path path ω as its history, then it will be in state s in the next step with probability $\mu(s)$, where $\sigma(\omega) = (d,a,\mu)$. We denote the set of infinite paths induced by a given scheduler σ to be $Path^{\sigma}_{ful} = \{\omega \in Path_{ful} \mid \sigma(\omega{\downarrow_i}) = (d_i,a_i,\mu_i)\ for\ i \geq 0\}$, where $\omega{\downarrow_i}$ returns the prefix of ω up to length i. Then we define $Path^{\sigma}_{ful}(s) = Path^{\sigma}_{ful} \cap Path_{ful}(s)$. Furthermore, for each $s \in S$ and scheduler σ, we can define the probability measure $Prob^{\sigma}_{s}$ over measurable sets of paths in the standard way [KSK76].

Given an infinite path $\omega = s_0 \xrightarrow{d_0,a_0,\mu_0} s_1 \xrightarrow{d_1,a_1,\mu_1} s_2 \xrightarrow{d_2,a_2,\mu_2} \cdots$, we let the *time-abstract trace* of ω be the infinite sequence of alternating state and action labels $L(s_0)a_0L(s_1)a_1L(s_2)a_2\cdots$. Given a scheduler σ, we let $\mathsf{trace}^{\sigma} : Path^{\sigma}_{ful} \to (2^{AP} \times Act)^{\omega}$ be the function associating the time-abstract trace with each path of σ. Then the *time-abstract trace distribution* of σ and state $s \in S$ is the probability measure over traces denoted by td^{σ}_{s} defined according to trace^{σ} and the trace distribution construction of Segala [Seg95]. Although we do not consider the details of the construction of trace distributions in this paper, we note that, for example, the probability assigned by td^{σ}_{s} to traces along which a certain atomic proposition a is exhibited is defined to be the same as the probability assigned by $Prob^{\sigma}_{s}$ to the set of paths along which a is exhibited. The set of time-abstract trace distributions of a TPS T is denoted by $\mathsf{tdist}(\mathsf{T}) = \{\mathsf{td}^{\sigma}_{s} \mid \sigma \text{ is a scheduler of } \mathsf{T} \text{ and } s \in S_0\}$.

2.2 Syntax of Probabilistic Parametric Timed Automata

We now extend the definition of probabilistic timed automata [KNSW07] to the parametric case. We assume a fixed set $X = \{x_1,\ldots,x_H\}$ of *clock variables*. A *clock variable* is a variable x_i which takes values in $\mathbb{R}_{\geq 0}$. All clocks evolve linearly at the same rate. We define a *clock valuation* as a function $w : X \to \mathbb{R}_{\geq 0}$ assigning a non-negative real value to each clock variable. We will often identify a valuation w with the point $(w(x_1),\ldots,w(x_H)) \in \mathbb{R}^{H}_{\geq 0}$. For $d \in \mathbb{R}_{\geq 0}$, we write $w+d$ to denote the valuation such that $(w+d)(x) = w(x)+d$ for all $x \in X$. Given a clock valuation w and a set $\rho \subseteq X$ of clocks, we denote by $\rho(w)$ the clock valuation such that $\rho(w)(x) = 0$ if $x \in \rho$ and $\rho(w)(x) = w(x)$ otherwise.

We also assume a fixed set $P = \{p_1,\ldots,p_M\}$ of *parameters*. A *parameter valuation* π is a function $\pi : P \to \mathbb{R}_{\geq 0}$ assigning a non-negative real value to each parameter. We will often identify a valuation π with the point $(\pi(p_1),\ldots,\pi(p_M)) \in \mathbb{R}^{M}_{\geq 0}$. A *linear inequality on the parameters P* (*linear inequality on the clock variables X and the parameters P*, respectively) is an inequality $e \prec e'$, where $\prec \in \{<,\leq\}$, and e,e' are two linear terms of the form:

$$\sum_i \alpha_i p_i + d, \quad (\sum_i \alpha_i p_i + \sum_j \beta_j x_j + d, \text{respectively})$$

where $1 \leq i \leq M, 1 \leq j \leq H$ and $\alpha_i,\beta_j,d \in \mathbb{N}$. A *constraint on the parameters P* (*constraint on the clock variables X and the parameters P*, respectively) is a conjunction of inequalities on P (on X and P, respectively).

In the sequel, the letter K (C, respectively) denotes a constraint on the parameters (on the clocks and the parameters, respectively). We consider `true` as a constraint on P, corresponding to the set of all possible values for P.

Given a parameter valuation π and a constraint C, we denote by $C[\pi]$ the constraint obtained by replacing each parameter p in C with $\pi(p)$. Likewise, given a clock valuation w, we denote by $C[\pi][w]$ the expression obtained by replacing each clock variable x in $C[\pi]$ with $w(x)$. A clock valuation w *satisfies* $C[\pi]$, denoted by $w \models C[\pi]$, if $C[\pi][w]$ evaluates to true. We say that π *satisfies* C, denoted by $\pi \models C$, if the set of clock valuations that satisfy $C[\pi]$ is nonempty. Similarly, we say that π *satisfies* K, denoted by $\pi \models K$, if the expression obtained by replacing each parameter p in K with $\pi(p)$ evaluates to true.

The following definition is an extension of the class of probabilistic timed automata to the parametric case. Parametric probabilistic timed automata allow the use of parameters in place of constants within guards and invariants, and are based on parametric timed automata [AHV93]. A *parametric probabilistic timed automaton (PPTA)* $\mathscr{A}$ is a tuple of the form $\mathscr{A} = (\Sigma, Q, \overline{q}, X, P, \mathscr{L}, I, prob)$, where:

- Σ is a finite set of *actions*,
- Q is a finite set of *locations* with an *initial location* $\overline{q} \in Q$,
- X is a finite set of clocks,
- P is a finite set of parameters,
- $\mathscr{L} : Q \to 2^{AP}$ is a labeling function for the locations,
- I is the *invariant* function, assigning to every $q \in Q$ a constraint $I(q)$ on the clocks X and the parameters P, and
- *prob* is the *probabilistic edge relation* consisting of elements of the form (q, g, a, η), where $q \in Q$, g is a constraint on the clocks X and the parameters P, $a \in \Sigma$, and $\eta \in \mathsf{Dist}(2^X \times Q)$.

We make the following assumptions on PPTA.

Determinism on action labels Given a location $q \in Q$ and action $a \in \Sigma$, there is at most one probabilistic edge of the form $(q, \_, a, \_) \in prob$.

Reset unicity For any probabilistic edge $(q, g, a, \eta) \in prob$ and location $q' \in Q$, there exists at most one $\rho \in 2^X$ such that $\eta(\rho, q') > 0$.

Neither of these assumptions is restrictive, because a PPTA non satisfying the assumptions can be transformed into a PPTA which does: for determinism on action labels, it is necessary to add and rename action labels, whereas, for reset unicity, it suffices to add an extra clock and additional locations. The assumptions of determinism on action labels and reset unicity are commonly met in practice, and they simplify the proofs of our subsequent results.

Let $\mathscr{A}$ be a PPTA. If, for each location $q \in Q$, we have that $I(q)$ is a constraint only on clocks, and, for each edge $(q, g, a, \eta) \in prob$, we have that g is a constraint only on clocks, we say that $\mathscr{A}$ is a *probabilistic timed automaton* (PTA).

2.3 Semantics of Probabilistic Parametric Timed Automata

In this section, we will consider the PPTA $\mathscr{A} = (\Sigma, Q, \overline{q}, X, P, \mathscr{L}, I, prob)$. Given a parameter valuation $\pi = (\pi_1, \ldots, \pi_M)$, we denote by $\mathscr{A}[\pi]$ the PTA obtained from $\mathscr{A}$ by substituting every occurrence of a parameter p_i by π_i in the guards and invariants. Formally, $\mathscr{A}[\pi] = (\Sigma, Q, \overline{q}, X, P, \mathscr{L}, I', prob')$, where I' and $prob'$ are defined in the following way: for each location $q \in Q$, we let $I'(q) = I(q)[\pi]$, and we let $prob'$ be the smallest set such that, for each $(q, g, a, \eta) \in prob$, we have $(q, g[\pi], a, \eta) \in prob'$.[1]

In the following, we consider the PTA $\mathscr{A}[\pi]$ from a given valuation π of the parameters. A *state* of $\mathscr{A}[\pi]$ is a pair $(q, w) \in Q \times \mathbb{R}^X_{\geq 0}$ such that $w \models I(q)[\pi]$. Informally, the behavior of $\mathscr{A}[\pi]$ can be understood as follows. The model starts in the initial location $\overline{q}$ with all clocks set to 0. In this, and any other state (q, w), there is a nondeterministic choice of either (1) making a *discrete (probabilistic) transition* or (2) letting *time pass*. In case (1), a discrete transition can be made according to any probabilistic edge $(q, g, a, \eta) \in prob$ with source location q which is *enabled*; that is the constraint g is satisfied by the current clock valuation w. Then the probability of moving to the location q' and resetting all of the clocks in ρ to 0 is given by $\eta(\rho, q')$. In case (2), the option of letting time pass is available only if the invariant $I(q)$ is satisfied while time elapses.

Formally, we define the semantics of a PTA as an associated infinite-state, infinite-branching TPS, defined as follows. The *TPS* (or *semantics*) *associated with* $\mathscr{A}[\pi]$ is $\mathsf{T}_{\mathscr{A}[\pi]} = (S, S_0, \Sigma, L, \Rightarrow)$ with $S = \{(q, w) \in Q \times (X \to \mathbb{R}_{\geq 0}) \mid w \models I(q)[\pi]\}$, $S_0 = \{(\overline{q}, \mathbf{0})\}$ where $\mathbf{0}(x) = 0$ for all $x \in X$, and where $((q, w), d, a, \mu) \in \Rightarrow$ if both of the following conditions hold :

Time elapse: $w + d \models I(q)[\pi]$;

Edge traversal: there exists a probabilistic edge $(q, g, a, \eta) \in prob$ such that $w + d \models g[\pi]$ and, for each $(\rho, q') \in \mathsf{support}(\eta)$, we have $\mu(q', \rho(w + d)) = \eta(\rho, q')$.

Finally, $L(q, w) = \mathscr{L}(q)$ for all $(q, w) \in S$. Observe that the rule for discrete transitions is a simplified version of the standard rule [KNSS02], and relies on the assumption of reset unicity. The definition of $\mathsf{T}_{\mathscr{A}[\pi]}$ relies on the fact that $\mathscr{A}$ and π satisfy the following assumptions:

Well-formedness: We say that a probabilistic timed automaton is *well-formed* if whenever a probabilistic edge is enabled it can be taken, i.e.: all of the probabilistic alternatives (pairs of target location and clock reset) result in states. Formally, a probabilistic timed automaton is said to be well-formed if, for each probabilistic edge $(q, g, a, \eta) \in prob$ and state $(q, w) \in S$ such that $w \models g[\pi]$, we require that $(q', \rho(w)) \in S$ for each $(\rho, q') \in \mathsf{support}(\eta)$.

A probabilistic timed automaton can be transformed into a well-formed probabilistic timed automaton by incorporating the invariant associated to the target location into the guard of each probabilistic edge (see [KNSW07]). Since this transformation has no effect on the semantics of the automaton, for the remainder of the paper we assume all probabilistic timed automata we consider are well-formed.[2]

[1] Strictly speaking, $\mathscr{A}[\pi]$ is a PTA only when π assigns a natural number (rather than a real) to each parameter, but this does not matter in our context.

[2] A counter-example of well-formed PTA is the following: let (q, w) be a state where $w(x) = 2$, let $(q, x \leq 2, a, \eta)$ be

No deadlock: To guarantee the existence of at least one transition from each state, we assume that $\mathscr{A}[\pi_0]$ has *no deadlock*, i.e.: in all states of $\mathscr{A}[\pi_0]$ reachable from $(\overline{q},\mathbf{0})$ (i.e., states in $Path_{fin}^{\mathsf{T}_{\mathscr{A}[\pi_0]}}$), it is always possible to take some probabilistic edge, possibly after letting time elapse [Spr01, JLS08]. This assumption guarantees that the probabilistic transition relation of the associated probabilistic system is total (see Section 2.1).

Note that a (P)PTA $\mathscr{A}$ for which all probabilistic edges feature point distributions can be interpreted as a (parametric) timed automaton. More precisely, the (parametric) timed automaton differs from $\mathscr{A}$ only in the edge relation: we represent a probabilistic edge $(q,g,a,\eta) \in prob$ of $\mathscr{A}$, for which $\eta(\rho,q') = 1$ for some $\rho \subseteq X$ and $q' \in Q$ (recall that $\mathscr{A}$ features point distributions only). We denote by $\eta = \mu_{(\rho,q')}$ the point distribution assigning probability 1 to (ρ,q')) for some $(\rho,q') \in 2^X \times Q$.

Networks of PPTA can be defined by using parallel composition based on the synchronization of discrete transitions of different components sharing the same action label in a similar manner to networks of PTA [KNS03]. For the sake of simplicity, we will suppose that synchronized actions are non-probabilistic (their distribution is a point distribution).

Finally, we lift some notation used for TPSs to PPTA: for example, we write $Path_{ful}^{\mathscr{A}[\pi]}$ for $Path_{ful}^{\mathsf{T}_{\mathscr{A}[\pi]}}$. We say that $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi']$ are *trace distribution equivalent*, written $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi']$, if $\mathsf{tdist}(\mathsf{T}_{\mathscr{A}[\pi]}) = \mathsf{tdist}(\mathsf{T}_{\mathscr{A}[\pi']})$. If $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi']$, we can conclude that the TPSs have time-abstract equivalent behaviors: for example, they assign the same maximum and minimum probabilities to the event of reaching a state labeled with the same set of atomic propositions, or of performing the same action [KNS02] (in general, they assign the same maximum and minimum probabilities to linear-time properties).

Remark. To rule out time-convergent behavior, it is classical to require *structural non-Zenoness*, i.e: we require that all structural loops in the graph of the PPTA must feature a reset of some clock which is forced to have the value of at least 1 by a lower bound of a guard in the loop [TYB05, JLS08]. This requirement is not necessary in our context, although it is natural to add it as an extra assumption.

3 Analysis of PPTAs Using the Inverse Method

In this section we consider an application of the inverse method to PPTAs. Our approach consists in applying the inverse method to a *non-probabilistic version* of the PPTA. The constraint output by the inverse method is also a solution to the inverse problem for the PPTA and the reference instantiation. Our ultimate goal is to generate the *weakest* constraint (i.e., satisfied by as many valuations as possible).

We first present formally the problem we intend to resolve, then introduce a method for obtaining (non-probabilistic) parametric timed automata from PPTAs. Finally we explain how the

a probabilistic edge such that $\eta(q',\emptyset) = \frac{1}{2}$ and $\eta(q'',\emptyset) = \frac{1}{2}$, and let $inv(q') = (x \leq 1)$ and $inv(q'') = (x \leq 2)$. Then the invariant of q' is not satisfied when taking the probabilistic edge $(q,x \leq 2,a,\eta)$, followed by the probabilistic selection of $(q',\emptyset)$, from (q,w).

results on the inverse method applied to parametric timed automata can be used to infer constraints on parameters of PPTAs.

3.1 The Inverse Problem on PPTAs

Consider the PPTA $\mathscr{A} = (\Sigma, Q, \overline{q}, X, P, \mathscr{L}, I, prob)$, which we assume is fixed throughout this section. Let π be a valuation of parameters in P, and let $((q,w),d,a,\mu) \in \Rightarrow$ be a transition of $\mathsf{T}_{\mathscr{A}[\pi]}$. Recall that, by reset unicity and the definition of $\mathsf{T}_{\mathscr{A}[\pi]}$, for each distinct $(q,w),(q',w') \in \mathsf{support}(\mu)$, we have $q \neq q'$. We define the distribution $\mathsf{loc}(\mu) \in \mathsf{Dist}(Q)$ over locations in the following way: for each $(q,w) \in S$, we let $\mathsf{loc}(\mu)(q) = \mu(q,w)$.

Let π' be a valuation of parameters in P. The infinite path $\omega = (q_0,w_0) \xrightarrow{d_0,a_0,\mu_0} (q_1,w_1) \xrightarrow{d_1,a_1,\mu_1} \cdots$ of $\mathsf{T}_{\mathscr{A}[\pi]}$, is *time-abstract path equivalent* to the infinite path $\omega' = (q'_0,w'_0) \xrightarrow{d'_0,a'_0,\mu'_0} (q'_1,w'_1) \xrightarrow{d'_1,a'_1,\mu'_1} \cdots$ of $\mathsf{T}_{\mathscr{A}[\pi]}$, written $\omega \equiv^{\mathsf{path}} \omega'$, if $q_i = q'_i$, $a_i = a'_i$, and $\mathsf{loc}(\mu_i) = \mathsf{loc}(\mu'_i)$ for all $i \in \mathbb{N}$. We extend the notion of time-abstract path equivalence to sets of paths: two sets Ω and Ω' of paths are time-abstract path equivalent, written $\Omega \equiv^{\mathsf{path}} \Omega'$, if (1) for each infinite path $\omega \in \Omega$, there exists $\omega' \in \Omega'$ such that $\omega \equiv^{\mathsf{path}} \omega'$, and (2) conversely, for each infinite path $\omega \in \Omega'$, there exists $\omega' \in \Omega$ such that $\omega \equiv^{\mathsf{path}} \omega'$.

Now consider two instantiations π and π' of P. We say that $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi']$ are *time-abstract scheduler equivalent*, written $\mathscr{A}[\pi] \equiv^{\mathsf{sched}} \mathscr{A}[\pi']$, if (1) for each scheduler σ of $\mathscr{A}[\pi]$ there exists a scheduler σ' of $\mathscr{A}[\pi']$ such that $Path^{\sigma}_{ful}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path^{\sigma'}_{ful}(\overline{q},\mathbf{0})$, and (2) for each scheduler σ of $\mathscr{A}[\pi']$ there exists a scheduler σ' of $\mathscr{A}[\pi]$ such that $Path^{\sigma}_{ful}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path^{\sigma'}_{ful}(\overline{q},\mathbf{0})$.

Next we recall results from [KNS02, KNS03] which allow to relate time-abstract equivalence on paths and schedulers to time-abstract trace distribution equivalence. The following proposition states that time-abstract path equivalence implies also time-abstract scheduler equivalence: this follows from the fact that counterparts of the paths of a scheduler of $\mathscr{A}[\pi]$ can be found in the set of paths of $\mathscr{A}[\pi']$, and themselves form the path set of a scheduler of $\mathscr{A}[\pi']$, and vice versa.

Proposition 1 *Let $\mathscr{A}$ be a (well-formed) PPTA, and let π and π' be instantiations of parameters P. If $Path^{\mathscr{A}[\pi]}_{ful}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path^{\mathscr{A}[\pi']}_{ful}(\overline{q},\mathbf{0})$, then $\mathscr{A}[\pi] \equiv^{\mathsf{sched}} \mathscr{A}[\pi']$.*

The next proposition follows from the fact that, by the requirement on distributions in the definition of time-abstract path equivalence, it is guaranteed that the branching structure of the probabilistic choices is the same for different schedulers with time-abstract path equivalent path sets.

Proposition 2 *Let $\mathscr{A}$ be a (well-formed) PPTA, and let π and π' be instantiations of parameters P. If $\mathscr{A}[\pi] \equiv^{\mathsf{sched}} \mathscr{A}[\pi']$, then $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi']$.*

In this paper, starting from an instantiation π_0 of the set P of parameters, we are interested in finding a constraint K_0 on the parameters such that $\pi_0 \models K_0$, and for any valuation π of P satisfying K_0, we have time-abstract path equivalence between the paths of $\mathscr{A}[\pi_0]$ and $\mathscr{A}[\pi]$. Furthermore, we suppose that $\mathscr{A}[\pi_0]$ has no deadlock. The problem can be stated as follows.

Consider a (well-formed) PPTA $\mathscr{A}$ and a valuation π_0 of the parameters such that $\mathscr{A}[\pi_0]$ has no deadlock. Find a constraint K_0 such that:

1. $\pi_0 \models K_0$,
2. $\mathscr{A}[\pi]$ has no deadlock, for all $\pi \models K_0$, and
3. $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi_0]$, for all $\pi \models K_0$.

The interest of finding such a constraint K_0, is that, by Propositions 1 and 2, the TPS associated to valuation π, for any $\pi \models K_0$, is ensured to have the same time-abstract behavior as for π_0.

3.2 Non-probabilistic Version of a PPTA

In this subsection, we state formal properties relating a (well-formed) PPTA to its non-probabilistic version. As explained in Section 2.3, it is straightforward to obtain a non-probabilistic parametric timed automaton from a PPTA featuring point distributions only. Given a PPTA $\mathscr{A}$, an *edge generated from* $(q,g,a,\eta) \in prob$ is a tuple (q,g,a,η,ρ,q') such that $\eta(\rho,q') > 0$. Let $\mathtt{edges}(q,g,a,\eta)$ be the set of the edges generated from (q,g,a,η), and let $\mathtt{edges} = \bigcup_{(q,g,a,\eta)\in prob} \mathtt{edges}(q,g,a,\eta)$ denote the set of all edges of $\mathscr{A}$.

The *non-probabilistic version* of $\mathscr{A}$, written $\mathscr{A}^*$, is a PPTA which agrees with $\mathscr{A}$ on all elements apart from the probabilistic edge relation. Formally, let $\mathscr{A}^* = (\Sigma, Q, \overline{q}, X, P, \mathscr{L}, I, prob^*)$ be the PPTA for which $prob^*$ is the smallest probabilistic edge relation such that for every edge $(q,g,a,\eta,\rho,q') \in \mathtt{edges}$, we have $(q,g,\langle\!\langle q,g,a,\eta,\rho,q'\rangle\!\rangle,\eta_{(\rho,q')}) \in prob^*$ (recall that $\eta_{(\rho,q')}$ denotes the point distribution assigning probability 1 to the element (ρ,q')). Observe that the state sets of $\mathsf{T}_{\mathscr{A}[\pi]}$ and $\mathsf{T}_{\mathscr{A}^*[\pi]}$ are equal. As noted in Section 2, from a PPTA for which all probabilistic edges feature point distributions, we can obtain the corresponding parametric timed automaton.

In the following proposition, we use $\rightarrow$ to refer to a transition of the semantic TPS of $\mathscr{A}[\pi]$, and $\rightarrow_*$ to refer to a transition of the semantic TPS of $\mathscr{A}^*[\pi]$.

Proposition 3 *Let π be an instantiation of P and (q,w) be a state of $\mathsf{T}_{\mathscr{A}[\pi]}$ (and $\mathsf{T}_{\mathscr{A}^*[\pi]}$). For each step $(q,w) \xrightarrow{d,a,\mu} (q',w')$ of $\mathsf{T}_{\mathscr{A}[\pi]}$, there exists the step $(q,w) \xrightarrow{d,\langle\!\langle q,g,a,\eta,\rho,q'\rangle\!\rangle,\mu_{(q',w')}}_* (q',w')$ of $\mathsf{T}_{\mathscr{A}^*[\pi]}$, where $(q,g,a,\eta) \in prob$ is such that $\mu(q',w') = \eta(\rho,q')$, and hence $\mathsf{loc}(\mu)(q') = \eta(\rho,q')$. Conversely, for each step $(q,w) \xrightarrow{d,\langle\!\langle q,g,a,\eta,\rho,q'\rangle\!\rangle,\mu_{(q',w')}}_* (q',w')$ of $\mathsf{T}_{\mathscr{A}^*[\pi]}$, there exists the step $(q,w) \xrightarrow{d,a,\mu} (q',w')$ of $\mathsf{T}_{\mathscr{A}[\pi]}$ such that $\mu(q',w') = \eta(\rho,q')$, and hence $\mathsf{loc}(\mu)(q') = \eta(\rho,q')$.*

Proposition 3 allows us to obtain a one-to-one mapping between transitions of $\mathscr{A}[\pi]$ and $\mathscr{A}^*[\pi]$. By reasoning inductively, we can extend the proposition to obtain a one-to-one mapping between paths of $\mathscr{A}[\pi]$ and $\mathscr{A}^*[\pi]$. Note that the probability of the transitions of $\mathscr{A}[\pi]$ is encoded in the action labels of the associated transitions of $\mathscr{A}^*[\pi]$. This, together with the one-to-one mapping between paths of $\mathscr{A}[\pi]$ and $\mathscr{A}^*[\pi]$, implies that, for any pair ω_*, ω'_* of paths such that $\omega_* \in Path_{ful}^{\mathscr{A}^*[\pi]}(\overline{q},\mathbf{0})$, $\omega'_* \in Path_{ful}^{\mathscr{A}^*[\pi']}(\overline{q},\mathbf{0})$ and $\omega_* \equiv^{\mathsf{path}} \omega'_*$, we can generate the paths ω, ω', such that $\omega \in Path_{ful}^{\mathscr{A}[\pi]}(\overline{q},\mathbf{0})$, $\omega' \in Path_{ful}^{\mathscr{A}[\pi']}(\overline{q},\mathbf{0})$ and $\omega \equiv^{\mathsf{path}} \omega'$. Together, these facts allow us to show the following.

Proposition 4 *Let $\mathscr{A}$ be a PPTA, and let π and π' be instantiations of parameters P. If $Path_{ful}^{\mathscr{A}^*[\pi]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}^*[\pi']}(\overline{q},\mathbf{0})$, then $Path_{ful}^{\mathscr{A}[\pi]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}[\pi']}(\overline{q},\mathbf{0})$.*

Proposition 5 *Let $\mathscr{A}$ be a (well-formed) PPTA, and π an instantiation of parameters P. Then, $\mathscr{A}^*[\pi]$ has no deadlock iff $\mathscr{A}[\pi]$ has no deadlock.*

3.3 Resolution of the Inverse Problem for PPTAs

In [ACEF], we have presented a method which solves the inverse problem for (the special case of) non-probabilistic parametric timed automata. The inverse problem can be described formally in the following way: given a non-probabilistic parametric timed automaton $\mathscr{A}$ and a valuation π_0 of the parameters, find a constraint K_0 such that $\pi_0 \models K_0$ and $Path_{ful}^{\mathscr{A}[\pi_0]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}[\pi]}(\overline{q},\mathbf{0})$ for all $\pi \models K_0$. A brief explanation of the method is given in the appendix.

The following algorithm explains how to use the inverse method in the extended framework of PPTAs.

ALGORITHM *InverseMethodPPTA*$(\mathscr{A},\pi_0)$
Input $\mathscr{A}$: PPTA
π_0 : Valuation of the parameters
Output K_0 : Constraint on the parameters

1. Construct the non-probabilistic version $\mathscr{A}^*$ of $\mathscr{A}$.
2. Construct K_0 for $\mathscr{A}^*$ using the classical inverse method.

Theorem 1 *Given a (well-formed) PPTA $\mathscr{A}$ and a reference valuation π_0 such that $\mathscr{A}[\pi_0]$ has no deadlock, the constraint K_0 returned by InverseMethodPPTA$(\mathscr{A},\pi_0)$ solves the inverse problem, i.e., $\pi_0 \models K_0$, and, for all $\pi \models K_0$, $\mathscr{A}[\pi]$ has no deadlock, and $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi_0]$.*

Proof. From Proposition 4 we have that $Path_{ful}^{\mathscr{A}^*[\pi]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}^*[\pi']}(\overline{q},\mathbf{0})$ implies $Path_{ful}^{\mathscr{A}[\pi]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}[\pi']}(\overline{q},\mathbf{0})$; given that $Path_{ful}^{\mathscr{A}^*[\pi]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}^*[\pi_0]}(\overline{q},\mathbf{0})$ for all $\pi \models K_0$, we have that $Path_{ful}^{\mathscr{A}[\pi]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}[\pi_0]}(\overline{q},\mathbf{0})$ for all $\pi \models K_0$. From Proposition 1 and Proposition 2, we conclude that $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi_0]$ for all $\pi \models K_0$.

Furthermore, from the fact that $\mathscr{A}[\pi_0]$ has no deadlock, it follows that $\mathscr{A}^*[\pi_0]$ has no deadlock, by Proposition 5. Hence, for all $\pi \models K_0$, $\mathscr{A}^*[\pi]$ has no deadlock (since $Path_{ful}^{\mathscr{A}^*[\pi_0]}(\overline{q},\mathbf{0}) \equiv^{\mathsf{path}} Path_{ful}^{\mathscr{A}^*[\pi]}(\overline{q},\mathbf{0})$). Therefore $\mathscr{A}[\pi]$ has no deadlock, by Proposition 5. □

From the fact that $\mathscr{A}[\pi] \approx^{\mathsf{tdist}} \mathscr{A}[\pi_0]$ for all $\pi \models K_0$, it follows, for example, that the maximum probability of reaching a state with a certain label will be the same in $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$.

Remark. As in [ACEF], the constraint K_0 output by our method is not necessarily the *weakest* constraint satisfying the inverse problem, as defined in Section 3.1.

4 Application of the Inverse Method to PPTA: Case Studies

In this section, we show the interest of the inverse method in the context of three case studies. More precisely, we consider three protocols, each modeled as a PPTA, and each with an associated reference valuation π_0 taken from the associated standard. For each protocol:

1. Using the tool IMITATOR [And09], which implements the inverse method in the non-probabilistic framework, we generate a constraint K_0 for the *non-probabilistic* version of the protocol.

2. Using the probabilistic model-checking tool PRISM [HKNP06, wp], we obtain minimum/maximum reachability probabilities for various properties according to parameter valuations satisfying K_0.

In particular, we check empirically that the minimum/maximum reachability probabilities computed for a number of parameter valuations satisfying K_0 are the same as those for the reference valuation π_0 (as predicted by Theorem 1).[3]

4.1 CSMA/CD Protocol

We first apply our method to the CSMA/CD Protocol described in Section 1. We consider the three parameters λ, σ and *slot* as described in [KNSW07, wp]. The following instantiation π_0 of the parameters is the reference valuation taken from the IEEE standard 802.3 for 10 Mbps Ethernet: $\lambda = 808\mu s$, $slot = 52\mu s$ and $\sigma = 26\mu s$. As sketched in Section 1, the non-probabilistic parametric timed automaton $\mathscr{A}^*$ is obtained as follows: we compose the (non-probabilistic) medium of Fig. 1 with the non-probabilistic version of the sender, obtained by replacing in Fig. 2 the random choice $backoff := \mathbf{RAND}(bc)$ with a non-deterministic choice (i.e., a set of 2^{bc+1} transitions associated with assignments of the form $backoff := i$, for $i = 0, 1, 2, \ldots, 2^{bc+1} - 1$).

Applying IMITATOR to this non-probabilistic model and the reference valuation π_0, we obtain the following constraint:

$$K_0 : \quad \sigma < slot \quad \wedge \quad 15slot < \lambda < 16slot.$$

This constraint is such that $\mathscr{A}[\pi]$ and $\mathscr{A}[\pi_0]$ are trace-distribution equivalent, for any $\pi \models K_0$.

We now check that different valuations π satisfying K_0 lead to the same minimum probability for the following four properties, described also with their associated PRISM syntax:

- $Prop_i$, for $i \in \{0, 1, 2\}$: minimum probability that sender 1 transmits its message after exactly i collisions, i.e., $\texttt{Pmin} =?[\texttt{F}(\texttt{s}_1 = \texttt{done} \ \& \ \texttt{nbCol} = i)]$.

- $Prop_{\leq 3}$: minimum probability that sender 1 transmits its message with no more than 3 collisions, i.e., $\texttt{Pmin} =?[\texttt{F}(\texttt{s}_1 = \texttt{done} \ \& \ \texttt{nbCol} \leq 3)]$.

We apply PRISM to the system with the parameters set to different valuations (including π_0), satisfying or not the constraint K_0. The results are given in Table 1. For all $\pi \models K_0$ (i.e., π_1 to π_4),

[3] Note that we consider acyclic versions of those protocols (roughly speaking, by bounding the maximal number of collisions in Section 4.1 and Section 4.3, and by bounding the number of rounds in Section 4.2).

Name	λ	*slot*	σ	$\models K_0$	$Prop_0$	$Prop_1$	$Prop_2$	$Prop_{\leq 3}$	Same as π_0
π_0	808	52	26	yes	0	0.5	0.375	0.96875	-
π_1	404	26	13	yes	0	0.5	0.375	0.96875	yes
π_2	31	2	1	yes	0	0.5	0.375	0.96875	yes
π_3	47	3	2	yes	0	0.5	0.375	0.96875	yes
π_4	940	60	59	yes	0	0.5	0.375	0.96875	yes
π_5	940	60	60	no	0	0	0.1875	0.609375	no
π_6	832	52	26	no	0	0.5	0.375	0.96875	yes
π_7	52	52	26	no	0	0.5	0.375	0.96875	yes

Table 1: Minimum probability that sender 1 transmits its message

the probabilities remain the same. An observation is that, as soon as we violate the constraint K_0 by considering the limit case where $\sigma = slot$ (i.e., π_5), the probabilities are different. A further observation is that, even if the value of λ violates K_0 (i.e., π_6 and π_7), the probabilities can remain the same. Indeed, the constraint generated by our method is not necessarily maximal, i.e., we can find valuations π of P (e.g., π_6 and π_7) s.t. $\pi \not\models K_0$ but the values of the probabilities remain the same as for π_0.

4.2 IEEE 1394 Root Contention Protocol

This case study concerns the root contention protocol of the IEEE 1394 ("FireWire") High Performance Serial Bus. We consider the instantiation of the parameters given in [KNS03, wp]. This instantiation π_0 is as follows[4]: $rc\_fast\_max = 85ns$, $rc\_fast\_min = 76ns$, $rc\_slow\_max = 167ns$, $rc\_slow\_min = 159ns$, and $delay = 30ns$. Applying IMITATOR to a parametric (non-probabilistic) timed automaton version of this model and the reference valuation π_0, we obtain the following constraint:

$$K_0 : 2delay < rc\_fast\_min \wedge rc\_fast\_max + 2delay < rc\_slow\_min.$$

We now check that different valuations π satisfying K_0 lead to the same minimum probability for the following properties:

- $Prop_{\leq 3}$: minimum probability that a leader is elected after 3 rounds or less: $\texttt{Pmin} = ?[\texttt{F}((\texttt{nbRounds1} \leq 3) \,\&\, (((\texttt{s1} = 8)\ \&\ (\texttt{s2} = 7)) | ((\texttt{s1} = 7)\ \&\ (\texttt{s2} = 8))))]$.
- $Prop_{\leq 5}$: minimum probability that a leader is elected after 5 rounds or less: $\texttt{Pmin} = ?[\texttt{F}((\texttt{nbRounds1} \leq 5) \,\&\, (((\texttt{s1} = 8)\ \&\ (\texttt{s2} = 7)) | ((\texttt{s1} = 7)\ \&\ (\texttt{s2} = 8))))]$.

The results of the application of PRISM to different parameter valuations are given in Table 2 (some parameter names are abbreviated for reasons of space). We notice as expected that, for all $\pi \models K_0$, the probabilities remain the same. Further experiments (including π_6) show that, as soon as the value of the parameters violates K_0, the probabilities become different from π_0. We note that the parameter valuation π_0 results in a state space of size 693107, for which $Prop_{\leq 5}$ could be verified in time 319.512s, whereas π_2 results in a state space of size 1393, for which $Prop_{\leq 5}$ could be verified in time 0.781s.[5]

[4] Note that the model given on PRISM's webpage allows both values $30ns$ and $360ns$ for *delay*.
[5] Experiments were performed on an Intel Core 2 Duo with 2GB of RAM.

Name	rf_max	rf_min	rs_max	rs_min	$delay$	$\models K_0$	$Prop_{\leq 3}$	$Prop_{\leq 5}$	Same as π_0
π_0	85	76	167	159	30	yes	0.875	0.96875	-
π_1	40	35	80	75	15	yes	0.875	0.96875	yes
π_2	4	3	8	7	1	yes	0.875	0.96875	yes
π_3	85	61	167	159	30	yes	0.875	0.96875	yes
π_4	85	76	167	146	30	yes	0.875	0.96875	yes
π_5	85	76	167	159	36	yes	0.875	0.96875	yes
π_6	85	76	167	159	37	no	0.71875	0.841796875	no

Table 2: Minimum probability that a leader is elected within 3 or 5 rounds

4.3 IEEE 802.11 Wireless Local Area Network Protocol

We also applied our method to the IEEE 802.11 Wireless Local Area Network Protocol, considering the following instantiation π_0 of the parameters mentioned in [wp] after rescaling from [KNS02][6]:

$$ASLOTTIME = 1\mu s \quad DIFS = 2\mu s \quad VULN = 1\mu s \quad TTMAX = 315\mu s$$
$$TTMIN = 4\mu s \quad ACK\_TO = 6\mu s \quad ACK = 4\mu s \quad SIFS = 1\mu s$$

Taking a parametric timed automaton version of the model and the parameter valuation π_0 as input, the tool IMITATOR computes the following constraint K_0:

$$\begin{array}{lllll} & VULN > 0 & \wedge & SIFS > 0 & \wedge \quad ACK\_TO + DIFS < 15 ASLOTTIME \\ \wedge & DIFS > 0 & \wedge & ASLOTTIME > 0 & \wedge \quad TTMIN + DIFS \leq TTMAX \\ \wedge & ACK \leq 2DIFS & \wedge & DIFS < TTMIN & \wedge \quad ACK\_TO + DIFS \leq ACK + TTMIN \\ \wedge & SIFS < TTMIN & \wedge & TTMIN \geq ACK & \wedge \quad TTMIN \leq ACK\_TO \\ \wedge & VULN < ACK & & & \end{array}$$

We now check that different valuations π satisfying K_0 lead to the same maximum probability that either station's backoff counter reaches k, for $k = 1,2,3$, as considered in [KNS02]. The results of the application of PRISM are given in Table 3, where the properties are denoted by $Prop_{k=i}$ for $k = 1,2,3$. The parameters $p_1, p_2, \ldots, p_8$ stand for $ASLOTTIME$, $DIFS$, $VULN$, $TTMAX$, $TTMIN$, ACK_TO, ACK, $SIFS$ respectively. Note that the real timings originating from the IEEE 802.11 standard (viz., in μs, $ASLOTTIME = 50$, $DIFS = 128$, $VULN = 48$, $TTMAX = 15,717$, $TTMIN = 224$, $ACK\_TO = 300$, $ACK = 205$, $SIFS = 28$) satisfy themselves the constraint K_0. Our approach thus provides us with a justification of the abstraction done in [KNS02] consisting in reducing the time scale of the model. Also observe that, as expected from the form of K_0, the value of $Prob_{k=i}$ is insensitive to important variations of $TTMAX$, i.e. parameter p_4 (provided its value remains greater or equal to $TTMIN + TTDIFS$, i.e $p_4 \geq p_2 + p_5$). For example, $Prob_{k=i}$ is equal for π_0 and π_2, in spite of the fact that p_4 is changed from 315 to 6.

5 Final Remarks

In this paper we have shown that the inverse method presented in [ACEF] can be applied, not just to non-probabilistic parametric timed automata, but also to their probabilistic extension, for proving time-abstract properties. The method relies on the conversion of PPTAs to non-probabilistic

[6] Note that, due to rescaling, the model given on PRISM's webpage allows several values for the parameters, e.g., 2 and 3 for $DIFS$.

Name	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	$\models K_0$	$Prop_{k=1}$	$Prop_{k=2}$	$Prop_{k=3}$	Same as π_0
π_0	1	2	1	315	4	6	4	1	yes	1	0.183593	0.017032	-
π_1	1	2	1	150	4	6	4	1	yes	1	0.183593	0.017032	yes
π_2	1	2	1	6	4	6	4	1	yes	1	0.183593	0.017032	yes
π_3	1	2	1	315	10	12	4	1	yes	1	0.183593	0.017032	yes
π_4	1	2	1	12	10	12	4	1	yes	1	0.183593	0.017032	yes
π_5	2	4	2	630	8	12	8	2	yes	1	0.183593	0.017032	yes
π_6	2	4	2	315	8	10	7	2	yes	1	0.183593	0.017032	yes

Table 3: Maximum probability of either station's backoff counter reaching k

parametric timed automata, then on the application of the inverse method of [ACEF]. The method has been shown to be successful in obtaining smaller parameter values, which can be helpful in reducing the size of the integer-time semantic probabilistic timed automata models prior to model checking.

Since the constraint output by our method is not maximal in general (see the remark in Section 3.3), it is interesting to design methods for enlarging it. In particular, the incremental method sketched out in [ACEF] could also be used in the probabilistic framework.

Let us finally point out that, in [KNS02, KNS03, KNPS06], another class of properties, named "soft deadline properties", is treated: for example, the minimum probability of a station delivering a packet *within some deadline*. Such properties are not "time-abstract", and fall beyond the class of those considered here. We note that soft deadline properties can be reduced to time-abstract location reachability properties, but with the addition of constraints within the model: hence, the inverse problem must be solved on the modified model. We plan to explore this approach in future work.

Bibliography

[ACEF] É. André, Th. Chatain, E. Encrenaz, L. Fribourg. An Inverse Method for Parametric Timed Automata. *International Journal of Foundations of Computer Science*. To appear.

[AD94] R. Alur, D. L. Dill. A theory of timed automata. *TCS* 126(2):183–235, 1994. http://dx.doi.org/10.1016/0304-3975(94)90010-8

[AHV93] R. Alur, T. Henzinger, M. Vardi. Parametric real-time reasoning. In *Proc. STOC '93*. Pp. 592–601. ACM, 1993. http://doi.acm.org/10.1145/167088.167242

[And09] E. André. IMITATOR: a Tool for Synthesizing Constraints on Timing Bounds of Timed Automata. In *ICTAC'09*. LNCS. Springer, 2009. To appear.

[Bey01] D. Beyer. Improvements in BDD-Based Reachability Analysis of Timed Automata. In *Proc. FME'01*. LNCS 2021, pp. 313–343. Springer, 2001.

[BMT99] M. Bozga, O. Maler, S. Tripakis. Efficient Verification of Timed Automata Using Dense and Discrete Time Semantics. In *CHARME'99*. LNCS 1703. Springer, 1999.

[CDF+08] N. Chamseddine, M. Duflot, L. Fribourg, C. Picaronny, J. Sproston. Computing Expected Absorption Times for Parametric Determinate Probabilistic Timed Automata. In *Proc. QEST'08*. Pp. 254–263. IEEE, 2008.

[Daw04] C. Daws. Symbolic and parametric model checking of discrete-time markov Chains. In *Proc. ICTAC'04*. LNCS 3407, pp. 280–294. Springer, 2004.

[HKM08] T. Han, J.-P. Katoen, A. Mereacre. Approximate parameter synthesis for probabilistic time-bounded reachability. In *Proc. RTSS'08*. Pp. 173–182. IEEE, 2008.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proc. TACAS'06*. LNCS 3920. 2006.

[HMP92] T. Henzinger, Z. Manna, A. Pnueli. What Good Are Digital Clocks? In *Proc. ICALP'92*. LNCS 623, pp. 545–558. Springer, 1992.

[Jen96] H. E. Jensen. Model checking probabilistic real time systems. In *Proc. of the 7th Nordic Work. on Progr. Theory*. Chalmers Institute of Technology, 1996.

[JLS08] M. Jurdziński, F. Laroussinie, J. Sproston. Model Checking Probabilistic Timed Automata with One or Two Clocks. *Logical Methods in Computer Science* 4(3), 2008.

[KNPS06] M. Kwiatkowska, G. Norman, D. Parker, J. Sproston. Performance Analysis of Probabilistic Timed Automata using Digital Clocks. *FMSD* 29:33–78, 2006.

[KNS02] M. Kwiatkowska, G. Norman, J. Sproston. Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In *Proc. PAPM/PROBMIV'02*. LNCS 2399, pp. 169–187. Springer, 2002.

[KNS03] M. Kwiatkowska, G. Norman, J. Sproston. Probabilistic Model Checking of Deadline Properties in the IEEE 1394 FireWire Root Contention Protocol. *Formal Aspects of Computing* 14(3):295–318, 2003.

[KNSS02] M. Kwiatkowska, G. Norman, R. Segala, J. Sproston. Automatic Verification of Real-time Systems with Discrete Probability Distributions. *TCS* 282:101–150, 2002.

[KNSW07] M. Kwiatkowska, G. Norman, J. Sproston, F. Wang. Symbolic Model Checking for Probabilistic Timed Automata. *Information and Computation* 205(7):1027–1077, 2007.

[KSK76] J. G. Kemeny, J. L. Snell, A. W. Knapp. *Denumerable Markov Chains*. Graduate Texts in Mathematics. Springer, 2nd edition, 1976.

[LMT03] R. Lanotte, A. Maggiolo-Schettini, A. Troina. Weak Bisimulation for Probabilistic Timed Automata and Applications to Security. In *Proc. SEFM'03*. Pp. 34–43. IEEE, 2003.
citeseer.ist.psu.edu/lanotte03weak.html

[wp] PRISM. web page. `http://www.prismmodelchecker.org/`.

[Seg95] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

[Spr01] J. Sproston. *Model Checking for Probabilistic Timed and Hybrid Systems*. PhD thesis, School of Computer Science, University of Birmingham, 2001.

[TYB05] S. Tripakis, S. Yovine, A. Bouajjani. Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design* 26(3):267–292, 2005.

ALGORITHM *InverseMethod*($\mathscr{A}$, π_0)
Inputs $\mathscr{A}$: Parametric timed automaton of initial location $\overline{q}$
π_0 : Reference valuation of the parameters
Output K_0 : Constraint on the parameters
Variables i : Current iteration
S : Current set of symbolic states ($S = \bigcup_{j=0}^{i} Post^j_{\mathscr{A}(K)}(\{(\overline{q},K)\})$)
K : Current constraint on the parameters

$i := 0$; $K := True$; $S := \{(\overline{q}, True)\}$
DO
  DO UNTIL there are no π_0-incompatible states in S
    Select a π_0-incompatible state (q,C) of S (i.e., s.t. $\pi_0 \not\models C$)
    Select a π_0-incompatible J in C (i.e., s.t. $\pi_0 \not\models J$)
    $K := K \wedge \neg J$; $S := \bigcup_{j=0}^{i} Post^j_{\mathscr{A}(K)}(\{(\overline{q},K)\})$
  OD *%% S is π_0-compatible*
  IF $Post_{\mathscr{A}(K)}(S) = \emptyset$ **THEN RETURN** $K_0 := \bigcap_{(q,C)\in S}(\exists X : C)$ **FI**
  $i := i+1$; $S := S \cup Post_{\mathscr{A}(K)}(S)$ *%% $S = \bigcup_{j=0}^{i} Post^j_{\mathscr{A}(K)}(\{(\overline{q},K)\})$*
OD

Figure 3: Algorithm *InverseMethod*

Appendix: The Inverse Method

Given a (classical) parametric timed automaton $\mathscr{A}$ and a reference instantiation π_0 of parameters, the inverse method outputs a constraint K_0 such that:

1. $\pi_0 \models K_0$,
2. $Path^{\mathscr{A}[\pi_0]}_{ful} \equiv^{\mathsf{path}} Path^{\mathscr{A}[\pi]}_{ful}$, for all $\pi \models K_0$.

The algorithm *InverseMethod* can be summarized as follows. Starting with $K := True$, we iteratively compute a growing set of reachable symbolic states. A symbolic state of the system is a couple (q,C), where q is a location of $\mathscr{A}$, and C a constraint on the parameters[7]. When a π_0-*incompatible* state (q,C) is encountered (i.e., when $\pi_0 \not\models C$), K is refined as follows: a π_0-incompatible inequality J (i.e., such that $\pi_0 \not\models J$) is selected within C, and $\neg J$ is added to K. The procedure is then started again with this new K, and so on, until no new reachable state is computed (we focus here on *acyclic* systems: see [ACEF] for details).

A simplified version of algorithm *InverseMethod* is given in Fig. 3, where the clock variables have been disregarded for the sake of simplicity. We denote by $Post^i_{\mathscr{A}}(S)$ the set of symbolic states reachable from S in at most i steps of $\mathscr{A}$.

There is an implementation of this algorithm, called IMITATOR, which is written in Python, and makes use of HYTECH for the computation of the *Post* operation. The Python program contains about 1500 lines of code, and its writing took about 4 man-months of work (see [And09]).

[7] Strictly speaking, C is a constraint on the parameters *and* the clock variables, but the clock variables are omitted here for the sake of simplicity. See [ACEF] for more details.

A coinductive approach to verified exact real number computation

Ulrich Berger and Sion Lloyd

Swansea University, Wales, UK

Abstract: We present an approach to verified programs for exact real number computation that is based on inductive and coinductive definitions and program extraction from proofs. We informally discuss the theoretical background of this method and give examples of extracted programs implementing the translation between the representation by fast converging rational Cauchy sequences and the signed binary digit representations of real numbers.

Keywords: Proof theory, program extraction, exact real number computation, coinduction

1 Introduction

In current implementations of main stream programming languages real numbers are represented in floating point format, and computation on real numbers is done with respect to this representation. As is well-known, rounding errors in floating point arithmetic may occur, and inevitably do so, due to the limited precision of floating point numbers on the one hand and the infinitary character of the real numbers on the other hand. The problem with this is not so much the fact that these rounding errors occur already in relatively simple computations, but that the user has no control over them. As a simple example consider the function

```
f(x) = 1+x-(x^2)*(x+1)*((1/x)-(1/(x+1)))     -- Haskell code
```

and the computations

```
*Main> f(10^4) :: Float
2.834961
*Main> f(10^4) :: Double
1.0000000006384653
*Main> f(10^9) :: Double
-149.21128177642822
```

Which of these results can we trust? Actually, in all three cases the correct result is 1.0, which can be easily seen by applying elementary school algebra to the expression defining the function f. The situation we encounter here is typical: in order to estimate the accuracy of a floating point computation one needs, in principle, always a mathematical analysis of the numerical stability of the problem which can be arbitrarily difficult. Increasing the precision cannot replace such an analysis because one does not know by how much the precision needs to be increased in order to obtain a required number of correct digits. In view of safety critical applications of numeric computation, for example autopilot systems for aircrafts, these problems can no longer be neglected,

but require alternative approaches that have a sound mathematical and technological basis. Such approaches are currently promoted under the slogan "computing with exact real numbers". In exact real number computation results are not necessarily exact, but they are guaranteed to be correct with any accuracy prescribed by the user. This means the user has full control over errors. Of course, it still can happen that a given accuracy cannot be obtained due to limited resources in time and space, but it will never happen that a result is delivered without information about its accuracy.

A further essential requirement in exact real number computation is that the correctness of a program has to be *proven* (in a stringent mathematical sense) in order to justify the user's trust in it. The generation of provably correct programs in exact real number computation is the focus of this paper.

In traditional program verification one takes a program and applies a certain method to prove that it meets a given specification (see the seminal papers [Flo67, Hoa97, Dij75, Pnu77, Mil80] and systems supporting program verification, e.g. PVS [ORSS98], Isabelle [NPW02], Coq [Coq], KIV [BRS+00], ACL2 [KMM00], BLAST [Bez07]). Another approach is to develop or derive programs according to certain rules that preserve correctness, thus obtaining programs that are correct "by construction" [Dij97, Gri81, DJ78]. The method we are presenting can be seen as a rather radical instance of the latter approach. ¿From a formal constructive proof of a mathematical statement A we extract a program that "realises" A. In general, the statement A does not need to be related to programming, but in specific cases A may be viewed as a specification of a computational problem that is solved by the extracted program. By a constructive proof we mean a proof that does not make use of the law of excluded middle or equivalent principles. Constructive reasoning is being adopted in Intuitionism [vH67, Hey56, Tro73], Constructive Mathematics [BB85] and Constructive Type Theory [ML84], and is implemented in a number of interactive proof systems, for example, NuPrL [Con86], Coq [Coq], Minlog [BBS+98] and Agda [Agd]. The logical basis of program extraction via realisability was laid by Kleene [Kle45] and Kreisel [Kre59] (for proof-theoretic purposes). It is an instance of what is known in Computer Science as the "Curry-Howard correspondence" or "proofs-as-programs" paradigm which is applicable to constructive proofs in a wide range of areas. In this paper we concentrate on constructive real analysis and we illustrate the method by some simple yet non-trivial examples. We want to make the case that it is not only possible in principle, but also feasible in practice to extract interesting programs from proofs (see also [Sch09] for related work on program extraction in constructive analysis).

An important principle for definitions and proofs we are using is *coinduction*. A coinductive definition can be viewed set-theoretically as the largest fixed-points of a monotone set operator or category-theoretically as the final coalgebra of a functor. Recently, coinductive definitions, coalgebras and coinductive proofs have become very popular for describing concurrent systems and cryptographic protocols [BS07, JR97, Mos99, Rut00, HW03]. Also, much of the recent work on exact real number computation uses coinduction to verify real number algorithms w.r.t. the representation of real numbers by infinite streams of signed digits and similar representations [EH02, ME07, GNSW07, CD06, Ber07, BH09]. In our paper we go one step further and extract these algorithms from proofs about coinductive characterisations of real numbers.

2 Induction and coinduction

We give a brief introduction to coinduction and the dual principle of induction. We begin with the latter as it is more familiar.

Consider an operator $\Phi : \mathscr{P}(U) \to \mathscr{P}(U)$, where U is a set and $\mathscr{P}(U)$ is the powerset of U, and assume that Φ is monotone, i.e. if $X \subseteq Y \subseteq U$, then $\Phi(X) \subseteq \Phi(Y)$. Since $\mathscr{P}(U)$ is a complete lattice w.r.t. set inclusion Φ has a least fixed point $\mu\Phi$, according to the Knaster-Tarski Theorem. In fact, $\mu\Phi$ is the least Φ-closed subset of U where a set $X \subseteq U$ is called Φ-closed if $\Phi(X) \subseteq X$. Hence we have the *closure principle* for $\mu\Phi$

$$\Phi(\mu\Phi) \subseteq \mu\Phi$$

as well as the *induction principle*

$$\text{if } \Phi(X) \subseteq X \text{ then } \mu\Phi \subseteq X$$

for all subsets X of U (one often says "induction on $\mu\Phi$"). In many cases Φ has a definition of the form $\Phi(X) = \{u \in U | A(u) \vee B(X,u)\}$ where $A(u)$ does not depend on X. Then the inclusion $\Phi(X) \subseteq X$ is equivalent to $\forall u \in U\ [(A(u) \Rightarrow X(u)) \vee (B(X,u) \Rightarrow X(u))]$ and the implications $A(u) \Rightarrow X(u)$ and $B(X,u) \Rightarrow X(u)$ are called induction *base* and *step* respectively, $B(X,u)$ is called *induction hypothesis*.

It is easy to see that μ, considered as an operation on monotone operators, is itself monotone, i.e. if $\Phi(X) \subseteq \Psi(X)$ for all $X \subseteq U$, then $\mu\Phi \subseteq \mu\Psi$. Indeed, by the induction principle it suffices to show $\Phi(\mu\Psi) \subseteq \mu\Psi$. But $\Phi(\mu\Psi) \subseteq \Psi(\mu\Psi) \subseteq \mu\Psi$. From the monotonicity of μ one can infer the following *strong induction* principle

$$\text{if } \Phi(X \cap \mu\Phi) \subseteq X \text{ then } \mu\Phi \subseteq X$$

For the proof of strong induction we assume $\Phi(X \cap \mu\Phi) \subseteq X$ which can be rewritten as $\Psi(X) \subseteq X$ where $\Psi(X) := \Phi(X \cap \mu\Phi)$. Clearly Ψ is a monotone operator. Thus $\mu\Psi \subseteq X$. Hence it is enough to show $\mu\Phi \subseteq \mu\Psi$. We prove this by induction on $\mu\Phi$. By the monotonicity result above we have the reverse inclusion $\mu\Psi \subseteq \mu\Phi$. Hence, $\Phi(\mu\Psi) = \Phi(\mu\Psi \cap \mu\Phi) = \Psi(\mu\Psi) \subseteq \mu\Psi$.

Example 1 (Natural Numbers) *Let $\mathbb{R}$ be the set of real numbers and define $\Phi : \mathscr{P}(\mathbb{R}) \to \mathscr{P}(\mathbb{R})$ by $\Phi(X) := \{0\} \cup \{y+1 | y \in X\}$. Then $\mu\Phi = \mathbb{N} = \{0,1,2,\dots\}$. The closure principle for $\mathbb{N}$ is equivalent to $\mathbb{N}(0) \wedge \forall x\,(\mathbb{N}(x) \to \mathbb{N}(x+1))$ while the induction principle for $\mathbb{N}$ is equivalent to $(X(0) \wedge \forall x\,(X(x) \to X(x+1)) \to \forall x\,(\mathbb{N}(x) \to X(x))$. The strong induction principle is similar, but with the step formula $\forall x\,(X(x) \to X(x+1))$ weakened to $\forall x \in \mathbb{N}\,(X(x) \to X(x+1))$.*

Now we turn our attention to coinduction which is dual to induction. For the same reason a monotone operator Φ has a least fixed point it has a greatest fixed point $\nu\Phi$. It is the largest Φ-coclosed subset of U where a set $X \subseteq U$ is called Φ-coclosed if $X \subseteq \Phi(X)$. Consequently, we have *coclosure*: $\nu\Phi \subseteq \Phi(\nu\Phi)$, and *coinduction*: if $X \subseteq \Phi(X)$ then $X \subseteq \nu\Phi$. With a similar argument as for μ one can show that ν is monotone and deduce from that a *strong coinduction* principle: if $X \subseteq \Phi(X \cup \nu\Phi)$ then $X \subseteq \nu\Phi$. We will see examples of coinduction in Sect. 3.

3 Cauchy and signed digit representations of real numbers

The primary objects of study in this paper are the real numbers in the compact interval

$$\mathbb{I} := [-1,1] = \{x \in \mathbb{R} \mid |x| \leq 1\}$$

Since real numbers are per-se abstract objects, it is not possible to compute with them directly: one has to refer to a specific representation. Two common representations of real numbers $x \in \mathbb{I}$ are

(1) Cauchy sequences $(q_n)_{n\in\mathbb{N}}$ where $q_n \in \mathbb{I}$ is rational with $|x - q_n| \leq 2^{-n}$ for all $n \in \mathbb{N}$,

(2) power series $x = \sum_{i\in\mathbb{N}} d_i 2^{-(i+1)}$ where $d_i \in \mathrm{SD} := \{0,1,-1\}$ (signed digits).

We consider the problem of producing a translation between the two representations that is formally proven to be correct. Note that in a traditional approach a quite complex formal system is required to deal with this problem: We need sorts for reals, rationals, natural numbers, digits and infinite sequences, and, in addition to the usual arithmetic operations, coercion functions between these sorts. Furthermore, the system must be able to express the recursive or iterative definitions of the higher-order functions translating between the two representations. On the other hand, the approach we are proposing and demonstrating here requires only one sort for real numbers with the usual first-order axioms for a real closed fields as well as the possibility to formalise inductive and coinductive definitions that were described in Sect. 2. In the following, all individual variables (denoted by lower case letters) range over real numbers.

We model the Cauchy representation (1) by the formula $A(x) := \forall n \in \mathbb{N} A_n(x)$ where $\mathbb{N}$ is defined as in Example 1 in Sect. 2 and

$$A_n(x) := \exists q \in \mathbb{Q}\,(|x| \leq 1 \wedge |x - q| \leq 2^{-n})$$

Here $\mathbb{Q}$ defines the rational numbers as a subset of the real numbers in the usual way with help of the predicate $\mathbb{N}$. E.g. $\mathbb{Q}(x) := \exists m,n,k \in \mathbb{N}\,(k > 0 \wedge xk = m - n)$ [1]. The formula $A(x)$ replaces the Cauchy representation in the sense that from a constructive proof of it one can extract a program implementing an infinite sequence $(q_n)_n \in \mathbb{N}$ satisfying (1). Note that this infinite sequence is not present in the formula $A(x)$. Details of how this extraction works in general will be given in Sect. 4.

We model the signed digit representation (2) by a coinductive definition, motivated by the observation that if $x = \sum_{i\in\mathbb{N}} d_i 2^{-(i+1)}$, then $|x - d_0/2| \leq 1/2$ and $2x - d_0 = \sum_{i\in\mathbb{N}} d_{i+1} 2^{-(i+1)}$. Therefore, we set $\mathbb{I}_d(x) := |x - d_0/2| \leq 1/2$ and define C as the largest set (of real numbers) such that

$$\mathrm{C}(x) \Rightarrow \exists d \in \mathrm{SD}(\mathbb{I}_d(x) \wedge \mathrm{C}(2x - d))$$

More formally $\mathrm{C} := \nu\mathscr{J}$ where the set operator $\mathscr{J} : \mathscr{P}(\mathbb{R}) \to \mathscr{P}(\mathbb{R})$ is defined by $\mathscr{J}(X) := \{x \mid \exists d \in \mathrm{SD}(\mathbb{I}_d(x) \wedge X(2x - d))\}$. Now, the program extracted from a proof of $\mathrm{C}(x)$ will be an infinite stream of digits $d_i \in \mathrm{SD}$ such that (2) holds.

In order to extract a program that translates between the representations (1) and (2) it will be sufficient to prove constructively the equivalence of the formulas $\forall n \in \mathbb{N} A_n(x)$ and $\mathrm{C}(x)$.

[1] The bounded quantifiers we used are just shorthands: $\forall x \in X\, B(x)$ stands for $\forall x\,(X(x) \to B(x))$ and $\exists x \in X\, B(x)$ stands for $\exists x\,(X(x) \wedge B(x))$.

4 Program extraction: theory

In this section we briefly describe the program extraction process in general, giving explanations of the main ideas priority over complete and formal definitions and correctness proofs.

4.1 The programming language

The programming language that will be the target of the extraction process is a λ-calculus with constructors and pattern matching and (ML-)polymorphic recursive types. We let α range over type variables.

$$\text{Type} \ni \rho, \sigma ::= \alpha \mid \mathbf{1} \mid \rho + \sigma \mid \rho \times \sigma \mid \rho \to \sigma \mid \text{fix}\,\alpha\,.\,\rho$$

In the definition of terms we let x range over term variables and C over constructors. It is always assumed that a constructor is applied to the correct number of arguments as determined by its arity. We will only use the constructors Nil (nullary), Left, Right (unary), Pair (binary), and $\text{In}_{\text{fix}\,\alpha\,.\,\rho}$ (unary) for every fixed point type $\text{fix}\,\alpha\,.\,\rho$.

$$\text{Term} \ni M, N ::= x \mid C(\vec{M}) \mid \text{case}\,M\,\text{of}\{C_1(\vec{x_1}) \to N_1\,; \ldots ; C_n(\vec{x_n}) \to N_n\} \mid \lambda x.M \mid (MN) \mid \text{rec}\,x\,.\,M$$

In a pattern $C_i(\vec{x_i})$ of a case-term all variables in $\vec{x_i}$ must be different. The typing relation $\Gamma \vdash M : \rho$ is defined inductively as follows.

$$\Gamma, x : \rho \vdash x : \rho \qquad \Gamma \vdash \text{Nil} : \mathbf{1} \qquad \frac{\Gamma, x : \rho \vdash M : \rho}{\Gamma \vdash \text{rec}\,x\,.\,M : \rho}$$

$$\frac{\Gamma, x : \rho \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \rho \to \sigma} \qquad \frac{\Gamma \vdash M : \rho \to \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash MN : \sigma}$$

$$\frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \text{Pair}(M,N) : \rho \times \sigma} \qquad \frac{\Gamma \vdash M : \rho \times \sigma \qquad \Gamma, x_1 : \rho, x_2 : \sigma \vdash K : \tau}{\Gamma \vdash \text{case}\,M\,\text{of}\{\text{Pair}(x_1,x_2) \to K\} : \tau}$$

$$\frac{\Gamma \vdash M : \rho}{\Gamma \vdash \text{Left}(M) : \rho + \sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \text{Right}(M) : \rho + \sigma}$$

$$\frac{\Gamma \vdash M : \rho + \sigma \qquad \Gamma, x_1 : \rho \vdash L : \tau \qquad \Gamma, x_2 : \sigma \vdash R : \tau}{\Gamma \vdash \text{case}\,M\,\text{of}\{\text{Left}(x_1) \to L; \text{Right}(x_2) \to R\} : \tau}$$

Let $\rho = \rho(\vec{\alpha}) = \text{fix}\,\alpha\,.\,\rho_0(\alpha, \vec{\alpha})$:

$$\frac{\Gamma \vdash M : \rho_0(\rho(\vec{\sigma}), \vec{\sigma})}{\Gamma \vdash \text{In}_\rho(M) : \rho(\vec{\sigma})} \qquad \frac{\Gamma \vdash M : \rho(\vec{\sigma}) \qquad \Gamma, x : \rho_0(\rho(\vec{\sigma}), \vec{\sigma}) \vdash K : \tau}{\Gamma \vdash \text{case}\,M\,\text{of}\{\text{In}_\rho(x) \to K\} : \tau}$$

Equipped with a lazy semantics (which we do not describe due to lack of space) this system can be viewed almost literally as a fragment of Haskell. For example, a type $\text{fix}\,\alpha\,.\,\rho$ where $\rho = \rho(\alpha, \beta)$ has no free type variables other than α and β, can be modelled in Haskell by the data declaration `data FixRho beta = InFixRho (Rho FixRho beta)` assuming that `Rho alpha beta` models ρ. A term $\text{rec}\,x\,.\,M$ is modelled by `let {x = M} in x`. In fact, in Sect. 5 we will present the extracted programs as Haskell code (however, for the general considerations in the remainder of this Section the system above is more convenient). It is easy to see that this system is ML-polymorphic: if $\vdash M : \rho(\vec{\alpha})$, then $\vdash M : \rho(\vec{\sigma})$ for arbitrary types $\vec{\sigma}$.

4.2 The object language

The object language $\mathscr{L}$ which is used to formalise the proofs we want to extract programs from is a first-order language extended by predicate variables and the possibility to form least and greatest fixed points of strictly positive (and hence monotone) operators. *Terms*, $r,s,t\ldots$, are built from constants, first-order variables and function symbols as usual. *Formulas*, $A,B,C\ldots$, are $s=t$, $\mathscr{P}(\vec{t})$ where $\mathscr{P}$ is a predicate (predicates are defined below), $A\wedge B$, $A\vee B$, $A\to B$, $\forall xA$, $\exists xA$. A *predicate* is either a predicate constant P, or a predicate variable X, or a comprehension term $\{\vec{x}\mid A\}$ where A is a formula and $\vec{x}$ is a vector of first-order variables, or an inductive predicate $\mu X.\mathscr{P}$, or a coinductive predicate $\nu X.\mathscr{P}$ where $\mathscr{P}$ is a predicate of the same arity as the predicate variable X and which is *strictly positive* in X, i.e. X does not occur free in any premise of a subformula of $\mathscr{P}$ which is an implication. The application, $\mathscr{P}(\vec{t})$, of a predicate $\mathscr{P}$ to a list of terms $\vec{t}$ is a primitive syntactic construct, except when $\mathscr{P}$ is a comprehension term, $\mathscr{P}=\{\vec{x}\mid A\}$, in which case $\mathscr{P}(\vec{t})$ stands for $A[\vec{t}/\vec{x}]$. We will frequently use common abbreviations such as $\mathscr{P}\subseteq\mathscr{Q}:=\forall\vec{x}(\mathscr{P}(\vec{x})\to\mathscr{Q}(\vec{x}))$, $\{\vec{t}\mid A\}:=\{\vec{x}\mid\exists\vec{y}(\vec{x}=\vec{t}\wedge A)\}$ where $\vec{y}$ are the variables occurring free in A or t, $f(\vec{\mathscr{P}}):=\{f(\vec{x})\mid\mathscr{P}_1(x_1)\wedge\ldots\wedge\mathscr{P}_n(x_n)\}$, and so on.

The *proof rules* for $\mathscr{L}$ are the usual ones for intuitionistic predicate calculus with equality [Hey56, Kle45, Tro73], plus the axiom schemes for inductive and coinductive predicates that were discussed in Sect. 2. In addition we allow any axioms expressible by non-computational formulas that hold in the intended model. Falsity can be defined as $\bot:=\mu X.X$ where X is a 0-ary predicate variable (i.e. a propositional variable). ¿From the induction axiom for $\bot$ it follows immediately $\bot\to A$ for every formula A.

For our examples it will be sufficient to have only one sort for real numbers in $\mathscr{L}$ (a many-sorted language would be possible as well) together with the usual algebraic equations and inequations for the operations on real numbers.

4.3 Realisability

The first step in program extraction is to assign to every $\mathscr{L}$-formula A a type $\tau(A)$, the type of potential realisers of A. If A contains neither predicate variables nor the logical connective $\vee$ (disjunction), then we call it *non-computational* (otherwise *computational*) and set $\tau(A)=\mathbf{1}$ (= `()` in Haskell). Otherwise, $\tau(\mathscr{P}(\vec{t}))=\tau(\mathscr{P})$ (for predicates $\mathscr{P}$ the type $\tau(\mathscr{P})$ is defined below), $\tau(A\wedge B)=\tau(A)\times\tau(B)$, $\tau(A\vee B)=\tau(A)+\tau(B)$, $\tau(A\to B)=\tau(A)\to\tau(B)$, $\tau(\forall xA)=\tau(\exists xA)=\tau(A)$, For predicates $\mathscr{P}$ we define $\tau(\mathscr{P})$ by $\tau(X)=\alpha_X$ where α_X is a type variable assigned in a one-to-one fashion to the predicate variable X, $\tau(\{\vec{x}\mid A\})=\tau(A)$, and $\tau(\mu X.\mathscr{P})=\tau(\nu X.\mathscr{P})=\operatorname{fix}\alpha_X.\,\tau(\mathscr{P})$.

As one can see, the mapping τ wipes out all first-order content of a formula (first-order terms and quantifiers), hence the type $\tau(A)$ can be viewed as the "propositional skeleton" of the formula A. This is necessarily so, since the sorts in our first order language ($\mathbb{R}$ in our example in Sect. 3) have no counterpart in our programming language.

The next step is to define for every formula A and every program term M of type $\tau(A)$ what it means for M to *realise* A, formally $M\,\mathbf{r}\,A$. The latter is a formula in the language $\mathbf{r}(\mathscr{L})$ which is obtained by adding to $\mathscr{L}$ a sort for program terms and extending all other constructions concerning formulas and proofs mutatis mutandis. The $\mathbf{r}(\mathscr{L})$-formula $M\,\mathbf{r}\,A$ is in fact shorthand for

$\mathbf{r}(A)(M)$ where the $\mathbf{r}(\mathscr{L})$-predicate $\mathbf{r}(A)$ is defined by structural recursion on A, relative to a fixed one-to-one mapping from $\mathscr{L}$-predicate variables X to $\mathbf{r}(\mathscr{L})$-predicate variables $\tilde{X}$ with one extra argument place for program terms. If the formula A has the free predicate variables $X_1,\dots,X_n$, then the predicate $\mathbf{r}(A)$ has the free predicate variables $\tilde{X}_1,\dots,\tilde{X}_n$. Simultaneously with $\mathbf{r}(A)$ we define a predicate $\mathbf{r}(\mathscr{P})$ for every predicate $\mathscr{P}$, where $\mathbf{r}(\mathscr{P})$ has one extra argument place for program terms. If A is non-computational, then $\mathbf{r}(A) = \{() \mid A\}$. If $\mathscr{P}$ is non-computational, then $\mathbf{r}(\mathscr{P}) = \{((),\vec{x}) \mid \mathscr{P}(\vec{x})\}$. In all other cases: For a non-computational formula A we set $M\,\mathbf{r}\,A := M = \mathrm{Nil} \wedge A$, and

$$\begin{array}{rclcrcl}
\mathbf{r}(\mathscr{P}(\vec{t})) & = & \{x \mid \mathbf{r}(\mathscr{P})(x,\vec{t})\} & & \mathbf{r}(A \to B) & = & \{f \mid f(\mathbf{r}(A)) \subseteq \mathbf{r}(B)\} \\
\mathbf{r}(A \vee B) & = & \mathrm{inl}(\mathbf{r}(A)) \cup \mathrm{inl}(\mathbf{r}(B)) & & \mathbf{r}(A \wedge B) & = & \mathrm{Pair}(\mathbf{r}(A),\mathbf{r}(B)) \\
\mathbf{r}(\exists y A) & = & \{x \mid \exists y\,(\mathbf{r}(A)(x))\} & & \mathbf{r}(\forall y A) & = & \{x \mid \forall y\,(\mathbf{r}(A)(x))\} \\
\mathbf{r}(X) & = & \tilde{X} & & \mathbf{r}(\{\vec{y} \mid A\}) & = & \{(x,\vec{y}) \mid \mathbf{r}(A)(x)\} \\
\mathbf{r}(\mu X.\mathscr{P}) & = & \mu\tilde{X}.\mathbf{r}(\mathscr{P}) & & \mathbf{r}(\nu X.\mathscr{P}) & = & \nu\tilde{X}.\mathbf{r}(\mathscr{P})
\end{array}$$

We see that quantifiers and the quantified variable, although ignored by the program M and its type, of course do play a role in the definition of realisability, i.e. the *specification* of the program.

Finally, we sketch how to extract from a proof of a formula A a program term M realising A. Assuming the proof is given in a natural deduction system the extraction process is straightforward and follows in most cases the usual pattern of the Curry-Howard correspondence: Any non-computational axiom has the trivial program Nil as extracted program. The introduction and elimination rules for conjunction, disjunction and implication correspond to pairing, projection, injections into a disjoint sum, pattern matching, lambda-abstraction, and application, respectively. The $\forall$-introduction rule, $\forall$-elimination rule, and the $\exists$-introduction rule are ignored, i.e. the extracted program of the conclusion is identical to the one of the premise. The $\exists$-elimination rule corresponds to application, more precisely, if the proofs of the premises $\exists x A$ and $\forall x\,(A \to B)$ (where x is not free in B) have extracted programs $M : \tau(A)$ and $N : \tau(A) \to \tau(B)$, respectively, then the extracted program for the conclusion B is simply the application $MN : \tau(B)$. The extracted programs of closure, $\Phi(\mu\Phi) \subseteq \mu\Phi$, and induction, $(\Phi(X) \subseteq X) \Rightarrow \Phi(\mu\Phi) \subseteq X$, are $\mathbf{in}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda x.\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(x)$ and

$$\mathbf{it}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda s.\mathrm{rec}\,f\,.\,\lambda x.\mathrm{case}\,x\,\mathrm{of}\{\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(y) \to s(\mathbf{map}_{\alpha,\rho} f\,y)\}$$

where it is assumed that $\tau(\Phi(X)) = \rho(\alpha)$. The term $\mathbf{map}_{\alpha,\rho}$ has type $(\alpha \to \beta) \to \rho(\alpha) \to \rho(\beta)$ and can be defined by induction on $\rho(\alpha)$. For coclosure, $\nu\Phi \subseteq \Phi(\nu\Phi)$, and coinduction, $(X \subseteq \Phi(X)) \Rightarrow X \subseteq \Phi(\nu\Phi)$, the extracted programs are $\mathbf{out}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda x.\mathrm{case}\,x\,\mathrm{of}\{\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(y) \to y\}$ and

$$\mathbf{coit}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda s.\mathrm{rec}\,f\,.\,\lambda x.\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(\mathbf{map}_{\alpha,\rho} f\,(s x))$$

We have the typings

$$\begin{array}{ll}
\vdash \mathbf{in}_{\mathrm{fix}\,\alpha\,.\,\rho} : \rho(\mathrm{fix}\,\alpha\,.\,\rho) \to \mathrm{fix}\,\alpha\,.\,\rho & \vdash \mathbf{it}_{\mathrm{fix}\,\alpha\,.\,\rho} : (\rho(\alpha) \to \alpha) \to (\mathrm{fix}\,\alpha\,.\,\rho) \to \alpha \\
\vdash \mathbf{out}_{\mathrm{fix}\,\alpha\,.\,\rho} : (\mathrm{fix}\,\alpha\,.\,\rho) \to \rho(\mathrm{fix}\,\alpha\,.\,\rho) & \vdash \mathbf{coit}_{\mathrm{fix}\,\alpha\,.\,\rho} : (\alpha \to \rho(\alpha)) \to \alpha \to \mathrm{fix}\,\alpha\,.\,\rho
\end{array}$$

The Soundness Theorem, stating that the program extracted from a proof does indeed realise the proven formula, is shown in [Ber09b, BS]. Soundness Theorems for similar systems can be found in [Tat98] and Miranda-Perea [MP05].

5 Program extraction: applications

In this Section we apply the program extraction procedure described in Sect. 4 to a proof of the equivalence of the real number representations described in Sect. 3. Below we give a fairly detailed constructive proof of the equivalence which can be easily formalised in the object system described in Sect. 4.2, and from which we then extract the program.

Before we do that we discuss a simple example that demonstrates the fact that it is indeed crucial for program extraction that proofs are constructive and no axioms other than the axioms for inductive and coinductive definitions and true non-computational axioms are used. The formula $\forall x,y\,(x\le y\vee y>x)$, although true in the real numbers, must not be used as an axiom because it is computational. We cannot prove it constructively either, because otherwise we could extract a (closed) program $M:\mathbf{Boole}$ realising it, where $\mathbf{Boole}:=\mathbf{1}+\mathbf{1}$. This would mean that the formula, setting $\mathbf{t}:=\mathrm{Left}(\mathrm{Nil})$ and $\mathbf{f}:=\mathrm{Right}(\mathrm{Nil})$,

$$\forall x,y\,((M=\mathbf{t}\wedge x\le y)\vee(M=\mathbf{f}\wedge y>x))$$

holds. Since M is closed, either $M=\mathbf{t}$ or $M=\mathbf{f}$. In the first case it would follow $\forall x,y\,(x\le y)$ in the second case $\forall x,y\,(x>y)$ which are both false statements (similar unprovability results were among Kleene's and Kreisel's original motivations for studying realisability). Of course, we can prove constructively the relativised formula $\forall x,y\in\mathbb{N}\,(x\le y\vee y>x)$ (by induction). The extracted program is a decision procedure for the ordering of natural numbers. Since $\mathbf{Nat}:=\tau(\mathbb{N}(x))=\mathrm{fix}\,\alpha.\,\mathbf{1}+\alpha$ the program works with the unary representation of natural numbers, more precisely, its type is $\mathbf{Nat}\to\mathbf{Nat}\to\mathbf{Boole}$. Similarly, we can prove constructively $\forall x,y\in\mathbb{Q}\,(x\le y\vee y>x)$ and extract a program deciding the ordering on rational numbers.

A formula which we can safely assume as an axiom is $\forall x\,(A(x)\to\mathbb{I}(x))$. Clearly, this formula is true and it has the trivial realiser $\lambda f.\mathrm{Nil}$.

5.1 Proofs

The following Lemma takes care of some simple technicalities in the main proof.

Lemma 1 *(a)* $\mathrm{SD}\subseteq\mathbb{Q}$.

(b) $\forall q\in\mathbb{Q}\,\exists d\in\mathrm{SD}\,[q-1/4,q+1/4]\cap\mathbb{I}\subseteq\mathbb{I}_d$.

(c) $\forall p\in\mathbb{Q}\,\exists q\in\mathbb{Q}\cap\mathbb{I}\,\forall x\in\mathbb{I}\,|x-q|\le|x-p|$.

Proof. Part (a) is obvious. For part (b) we do a case analysis on the position of q. For $q<-\frac{1}{4}$ choose $d=-1$, for $-\frac{1}{4}\le q\le\frac{1}{4}$ choose $d=0$ and for $\frac{1}{4}<q$ choose $d=1$. It is easy to see that in all cases d is as required. For part (c) let $p\in\mathbb{Q}$ be given. Depending on whether $p<-1$ or $|p|\le 1$ or $p>1$ we set $q:=-1$ or $q:=p$ or $q:=1$. The required property obviously holds. □

Proposition 1 $\forall x\,(\mathrm{C}(x)\Leftrightarrow A(x))$.

Proof. For the implication from left to right we prove $\forall n\in\mathbb{N}(\mathrm{C}(x)\Rightarrow A_n(x))$ by induction on $\mathbb{N}$. Base $n=0$: If $\mathrm{C}(x)$, then clearly $x\in\mathbb{I}$, i.e. $|x|\le 2^0$. Hence we can take $q:=0$ to satisfy $A_0(x)$.

Step: The induction hypothesis is $\forall x(\mathrm{C}(x) \Rightarrow \exists q|x-q| \leq 2^{-n})$. We have to show $\forall x(\mathrm{C}(x) \Rightarrow \exists q|x-q| \leq 2^{-(n+1)})$. Assume $\mathrm{C}(x)$. By the coclosure principle for C we have $\mathbb{I}_d(x)$ and $\mathrm{C}(2x-d)$ for some $d \in \mathrm{SD}$. Set $x' := 2x-d$. By induction hypothesis, $|x'-q| \leq 2^{-n}$ for some q. Hence $\frac{|x'-q|}{2} \leq \frac{2^{-(n+1)}}{2}$, i.e. $\frac{|2x-d-q|}{2} \leq 2^{-(n+1)}$, i.e. $|x-\frac{d}{2}-\frac{q}{2}| \leq 2^{-(n+1)}$, i.e. $|x-\frac{d+q}{2}| \leq 2^{-(n+1)}$, so we may take $q' := \frac{d+q}{2}$ using Lemma 1 (a).

For the implication from right to left we need to show $A \subseteq \mathrm{C}$. By applying coinduction it is sufficient to show $A \subseteq \mathscr{J}(A)$, i.e.

$$\forall x(A(x)) \Rightarrow \exists d \in \mathrm{SD}\,(\mathbb{I}_d(x) \wedge A(2x-d)))$$

Assume $A(x)$. Then $\mathbb{I}(x)$ by the axiom discussed above. We have to find $d \in \mathrm{SD}$ such that

$$\mathbb{I}_d(x) \wedge \forall n \in \mathbb{N} \exists q \in \mathbb{Q} \cap \mathbb{I}\,|(2x-d)-q| \leq 2^{-n}$$

Using the assumption $A(x)$ with $n=2$, we obtain a $q \in \mathbb{Q} \cap \mathbb{I}$ such that $|x-q| \leq \frac{1}{4}$. According to Lemma 1 (b) there is some $d \in \mathrm{SD}$ such that $[q-1/4, q+1/4] \cap \mathbb{I} \subseteq \mathbb{I}_d$. Now let $n \in \mathbb{N}$. We have to find $q \in \mathbb{Q} \cap \mathbb{I}$ such that $|(2x-d)-q| \leq 2^{-n}$. We use the assumption $A(x)$ with $n+1$ and obtain $q' \in \mathbb{Q} \cap \mathbb{I}$ such that $|x-q'| \leq 2^{-(n+1)}$. Since $x \in \mathbb{I}_d$ and because of Lemma 1 (c) we may assume without loss of generality that $q' \in \mathbb{I}_d$. Hence $q := 2q'-d \in \mathbb{I}$ and $|(2x-d)-q| = 2|x-q'| \leq 2^{-n}$. □

5.2 Programs

We begin our program extraction by declaring the types of the predicates SD and $\mathbb{Q}$. We do not bother deriving them pedantically following Sect. 4, but define them in Haskell in a convenient way. For $\mathbb{Q}$ it is most convenient to use the build-in type of exact rational numbers.

```
data SD = N | Z | P deriving Show
type Rat = Rational
```

Next we extract programs from Lemma 1. Again, because the proofs are so simple, we do the extraction in an intuitive and non-pedantic way using build-in operations on the rational numbers. We also apply, here and in the following, some simplifications to types and extracted programs. For example, if B is computational, but A is not, we set $\tau(A \wedge B) = \tau(A \to B) = \tau(B)$ (instead of $\mathbf{1} \times \tau(B)$ and $\mathbf{1} \to \tau(B)$) and adjust realisability and program extraction accordingly.

```
lema :: SD -> Rat
lema = \d-> case d of {N -> -1; Z -> 0; P -> 1}

lemb :: Rat -> SD
lemb q | q < -1/4  = N
       | q > 1/4   = P
       | otherwise = Z

lemc :: Rat -> Rat
lemc q | q < -1    = -1
       | q > 1     = 1
       | otherwise = q
```

Now we declare the data type $\tau(\mathbb{N})$:

```
type NAT alpha = Either () alpha
data Nat = ConsNat (NAT Nat)
```

For later use we define the successor operation and numerals on `Nat`:

```
suc :: Nat -> Nat
suc n = ConsNat (Right n)

num :: Int -> Nat
num n = if n <= 0 then ConsNat (Left ()) else suc (n-1)
```

The program `suc` can be extracted from a proof that $\mathbb{N}$ is closed under successor. Now we move on to a more interesting part, the extracted programs of the axioms for the inductive predicate $\mathbb{N}$ and the coinductive predicate C. Below, `Sds` stands for "signed digit stream".

```
mapNAT :: (alpha -> beta) -> NAT alpha -> NAT beta
mapNAT = \f-> \z->
  case z of {Left u -> Left () ; Right x -> Right (f x)}

itNat :: (NAT alpha -> alpha) -> Nat -> alpha
itNat = \step->
 let {f = \n-> case n of {ConsNat z -> step (mapNAT f z)}} in f

type SDS alpha = (SD,alpha)
data Sds = ConsSds (SDS Sds)

mapSDS :: (alpha -> beta) -> SDS alpha -> SDS beta
mapSDS = \f-> \z-> case z of {(d,x) -> (d,f x)}

coitSds :: (alpha -> SDS alpha) -> alpha -> Sds
coitSds = \f-> \x-> ConsSds (mapSDS (coitSds f) (f x))
```

The type of the formula $A(x)$ is

```
type Approx = Nat -> Rat
```

Finally, the programs extracted from Prop. 1 are

```
proplr :: Sds -> Approx
proplr = \a-> \n-> itNat proplrStep n a

proplrStep :: NAT (Sds -> Rat) -> Sds -> Rat
proplrStep = \z-> \a-> case z of
 {Left u -> 0; Right ih -> case a of
                 {ConsSds (d,a') -> (lema d + ih a')/2}}
```

```
proprl :: Approx -> Sds
proprl = coitSds proprlStep

proprlStep :: Approx -> SDS Approx
proprlStep = \f->
  let d = lemb (f (num 2))
  in (d, \n-> lemc (2 * f (suc n) - lema d))
```

5.3 Results

As an example, we compute the signed digit representation of $\frac{1}{\sqrt{e}} \in \mathbb{I}$. Since

$$\frac{1}{\sqrt{e}} = e^{-\frac{1}{2}} = \sum_{i=0}^{\infty} \frac{(-\frac{1}{2})^i}{i!}$$

and the series converges at an exponential rate we define

$$e_n := \sum_{i=0}^{n} \frac{(-\frac{1}{2})^i}{i!} \in \mathbb{Q}$$

and obtain an infinite sequence $e = (e_n)_{n\in\mathbb{N}}$ realising the formula $A(\frac{1}{\sqrt{e}})$. Feeding e into the program `proplr` we obtain a realiser of $C(\frac{1}{\sqrt{e}})$, i.e. a signed digit representation of $\frac{1}{\sqrt{e}}$. In order to display the stream in the usual Haskell format we coerce `Sds` into `[SD]`.

```
e :: Approx
e n = sum [((-1/2)^i) / (fromIntegral (product [1..i]))
           | i <- [0..(fromIntegral n')]]
  where n' = nat2Int n

nat2Int :: Nat -> Int
nat2Int (ConsNat (Left ())) = 0
nat2Int (ConsNat (Right n)) = 1 + nat2Int n

sds2Stream :: Sds -> [SD]
sds2Stream (ConsSds (d,a)) = d : sds2Stream a

sde :: [SD]
sde = sds2Stream (proprl e)

*Main> take 100 sde
[P,Z,P,Z,N,P,Z,N,P,N,Z,Z,P,N,P,Z,N,Z,P,N,P,Z,Z,Z,Z,Z,N,Z,Z,P,
 Z,N,P,Z,Z,Z,Z,N,Z,P,N,P,Z,N,Z,Z,Z,Z,P,N,Z,P,Z,Z,Z,Z,Z,Z,Z,N,
 Z,P,Z,Z,Z,N,P,N,P,Z,N,Z,P,Z,Z,N,P,N,P,Z,N,P,N,Z,Z,P,Z,N,P,N,
 P,N,P,N,P,N,Z,P,Z,N]
```

We can check that this stream is correct by computing the corresponding floating point approximation and comparing it with the result obtained by floating point arithmetic (ironically, relying on the correctness of the latter).

```
list2Double :: [SD] -> Double
list2Double  = foldr av 0   where
    av d x = ((fromRational (lema d))+x)/2

*Main> exp (-0.5)
0.6065306597126334
*Main> list2Double (take 100 sde)
0.6065306597126334
```

6 Conclusion

We presented a method for extracting certified programs from constructive proofs. The method is based on a variant of realizability that strictly separates the (abstract) mathematical model from the data types the extracted program is dealing with. The latter are determined completely by the propositional structure of formulas and proofs. This has the advantage that the abstract mathematical structures do not need to be 'constructivised'. In addition, formulas that do not contain disjunctions are computationally meaningless and can therefore be taken as axioms as long as they are true. This enormously reduces the burden of formalization and turns - in our opinion - program extraction into a realistic method for the development of nontrivial certified algorithms.

We used the problem of translating between different representations of real numbers as an example to illustrate the method in general, and to show the use of inductive and coinductive definitions in program extraction.

Currently, we are working on an extension of this work to the situation where not only real numbers, but also real functions are coinductively represented and where the underlying domain is extended to arbitrary separable metric spaces or even more general structures [Ber09b, Ber09a].

An important aspect of the program extraction method is the ability to import existing certified software. This is partly accounted for by the possibility to add statements as axioms for which programs provably realising them are given. But as least as important is the possibility to include existing trusted data structures (large integers exact rationals, etc.) together with efficient (and certified) implementations of basic operations. It is possible to extend our approach in this respect, for example, using a general theory of realisability based on equilogical spaces and assemblies [BBS04].

The method of program extraction including inductive definitions is implemented for example in the Minlog system [BBS+98]. Carrying out in Minlog case studies involving coinduction as presented here requires an appropriate extension of the system. This is work in progress.

Bibliography

[Agd] Agda. http://wiki.portal.chalmers.se/agda/.

[BB85] E. Bishop, D. Bridges. *Constructive Analysis*. Grundlehren der mathematischen Wissenschaften 279. Springer, Berlin, Heidelberg, NewYork, Tokyo, 1985.

[BBS+98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, W. Zuber. Proof theory at work: Program development in the Minlog system. In Bibel and Schmitt (eds.), *Automated Deduction – A Basis for Applications*. Applied Logic Series II, pp. 41–71. Kluwer, Dordrecht, 1998.

[BBS04] A. Bauer, L. Birkedal, D. S. Scott. Equilogical spaces. *Theor. Comput. Sci.* 315(1):35–59, 2004.
http://dx.doi.org/10.1016/j.tcs.2003.11.012

[Ber07] Y. Bertot. Affine functions and series with co-inductive real numbers. *Math. Struct. Comput. Sci.* 17:37–63, 2007.

[Ber09a] U. Berger. From coinductive proofs to exact real arithmetic. 2009. To appear: LNCS Proceedings of Computer Science Logic (CSL), Coimbra, 7-11 September.

[Ber09b] U. Berger. Realisability for induction and coinduction. 2009. To appear: Proceedings of Computability and Complexity in Analysis (CCA), Ljubljana, Slovenia, 18-22 August, 2009.

[Bez07] M. Bezem. The Software Model Checker Blast. *Journal on Software Tools for Technology Transfer* 9(5-6):505–525, 2007.

[BH09] U. Berger, T. Hou. Coinduction for Exact Real Number Computation. Theory Comput. Sys., to appear, 2009.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, A. Programmiermethodik. Formal system development with KIV. In *Fundamental Approaches to Software Engineering, number 1783 in LNCS*. Pp. 363–366. Springer, 2000.

[BS] U. Berger, M. Seisenberger. Program extraction via typed realisability for induction and coinduction. Submitted.

[BS07] J. Bradfield, C. Stirling. Modal mu-calculi. In Blackburn et al. (eds.), *Handbook of Modal Logic*. Studies in Logic and Practical Reasoning 3, pp. 721–756. Elsevier, 2007.

[CD06] A. Ciaffaglione, P. Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theor. Comput. Sci.* 351:39–51, 2006.

[Con86] R. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice–Hall, New Jersey, 1986.

[Coq] The Coq Proof Assistant. http://coq.inria.fr/.

[Dij75] E. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Comm. ACM* 18:453–457, 1975.

[Dij97] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[DJ78] B. Dines, C. Jones. *The Vienna Development Method: The Meta-Language*. LNCS 61. Springer, Berlin, Heidelberg, New York, 1978.

[EH02] A. Edalat, R. Heckmann. Computing with Real Numbers: I. The LFT Approach to Real Number Computation; II. A Domain Framework for Computational Geometry. In Barthe et al. (eds.), *Applied Semantics - Lecture Notes from the International Summer School, Caminha, Portugal*. Pp. 193–267. Springer, 2002.

[Flo67] R. Floyd. Assigning meaning to Programs. In *Mathematical Aspects of Computer Science*. Pp. 19–32. American Mathematical Society, 1967.

[GNSW07] H. Geuvers, M. Niqui, B. Spitters, F. Wiedijk. Constructive analysis, types and exact real numbers. *Math. Struct. Comput. Sci.* 17(1):3–36, 2007.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[vH67] J. van Heijenoort (ed.). *From Frege to Gödel. A Source Book in Mathematical Logic 1879–1931*. Harvard University Press, Cambridge, MA., 1967. Reprinted 1970.

[Hey56] A. Heyting. *Intuitionism: An Introduction*. North-Holland, Amsterdam. Third Revised Edition 1971, 1956.

[Hoa97] T. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12:567–580, 1997.

[HW03] J. Hughes, M. Warnier. The Coinductive Approach to Verifying Cryptographic Protocols. In Wirsing et al. (eds.), *Recent Trends in Algebraic Development Techniques*. LNCS 2755, pp. 268–283. Springer, Berlin, 2003.

[JR97] B. Jacobs, J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin* 62:222–259, 1997.

[Kle45] S. C. Kleene. On the interpretation of intuitionistic number theory. *Jour. Symb. Logic* 10:109–124, 1945.

[KMM00] M. Kaufmann, P. Manolios, J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics*, pp. 101–128, 1959.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[ME07] J. R. Marcial-Romero, M. H. Escardo. Semantics of a sequential language for exact real-number computation. *Theor. Comput. Sci.* 379(1-2):120–141, 2007.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[MP05] F. Miranda-Perea. Realizability for Monotone Clausular (Co)Inductive Definitions. *Electronic Notes in Theoretical Computer Science* 123:179–193, 2005.

[Mos99] L. S. Moss. Coalgebraic Logic. *Annals of Pure and Applied Logic* 96, 1999.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[ORSS98] S. Owre, J. Rushby, N. Shankar, D. Stringer-Calvert. PVS: An Experience Report. In Hutter et al. (eds.), *Applied Formal Methods—FM-Trends 98*. Lecture Notes in Computer Science 1641, pp. 338–345. Springer-Verlag, Boppard, Germany, oct 1998.
http://www.csl.sri.com/papers/fmtrends98/

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *In Proc. 18th IEEE Symposium on Foundation of Computer Science*. LNCS 902, pp. 350–364. IEEE, 1977.

[Rut00] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249(1):3–80, 2000.

[Sch09] H. Schwichtenberg. Realizability interpretation of proofs in constructive analysis. Theory Comput. Sys., to appear, 2009.

[Tat98] M. Tatsuta. Realizability of Monotone Coinductive Definitions and Its Application to Program Synthesis. In Parikh (ed.), *Mathematics of Program Construction*. Lecture Notes in Mathematics 1422, pp. 338–364. Springer, 1998.

[Tro73] A. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Lecture Notes in Mathematics 344. Springer, 1973.

Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form

Loïc Besnard[1], Thierry Gautier[1], Matthieu Moy[2], Jean-Pierre Talpin[1], Kenneth Johnson[1] and Florence Maraninchi[2]

[1]INRIA Centre Rennes-Bretagne Atlantique/CNRS IRISA. Campus de Beaulieu. 35042 Rennes Cedex, FRANCE

[2]Verimag. Centre Équation. 2, avenue de Vignate. 38610 Gières, FRANCE

Abstract: We present an approach for the translation of imperative code (like C, C++) into the synchronous formalism SIGNAL, in order to use a model-checker to verify properties on the source code. The translation uses SSA as an intermediate formalism, and the GCC compiler as a front-end. The contributions of this paper with respect to previous work are a more efficient translation scheme, and the management of parallel code. It is applied successfully on simple SYSTEMC examples.

Keywords: SSA, synchronous, Signal, C, compilation, GCC, model-checking

1 Introduction

1.1 Context and motivations

Nowadays, embedded systems are becoming more and more complex, with stronger and stronger constraints of many kinds: cost, reliability, short life-cycle, and so on. Design correctness of software and hardware functionalities of embedded system is one of the major challenges and priorities for designers using software programming languages such as SYSTEMCand C/C++ to describe their systems. These programming languages allow for a comfortable design entry, fast simulation, and software/hardware co-design. Moreover, as the complexity of systems increases, designers are bound to reuse existing Intellectual Property components (IPs) in their design to improve the design productivity. However, system validation is a critical challenge for design reuse based on software programming languages. In recent years, many automated simulator and test tools have been developed to deal with design verification problems. However, mere simulation with non-formal development tools does by no means cover all design errors. What we therefore need is to use formal methods to ensure the quality of system designs. Among formal methods, model-checking [CGP99] has proved successful at increasing the reliability of some systems.

On the other hand, synchronous languages [BB91, BCE+03] have been introduced and used successfully for the design and implementation of real-time critical software. They rely on mathematical models such as data-flow equations or finite state machines that enable formal reasoning on designs. As a matter of fact, their associated toolsets provide among other formal transformations, automatic code generation, and verification of properties.

Relying on these bases, we propose an approach in which we automatically translate C/C++

models into the synchronous formalism SIGNAL [LGLL91, LTL03], hence enabling the application of formal methods without having to deal with the complex and error prone task to build formal models by hand. In particular, this allows one to use the SIGNAL toolbox, which includes a code generator, a model-checker...

We base our approach and associated tool on previous studies for the translation of imperative languages to the synchronous data-flow language SIGNAL [TLSG05, KTBB06]. This translation relies on the use of SSA ("Static Single Assignment") as intermediate representation of programs. Until now however, those previous works had not been seriously implemented nor experimented. Only manual experiments had been completed.

Moreover we extend the existing approach in two ways: first, the new translation scheme is more efficient, since it generates as few state variables as possible, thus reducing the work of the model-checker, and secondly, the tool now manages parallel, non-preemptive code.

As an example of such parallel language, we study the particular case of SYSTEMC, a library for C++ for the high-level modeling of Systems-on-a-Chip, which provides among other things a non-preemptive scheduler for C++ processes.

The translation specified here was implemented in the POLYCHRONY toolset [INRa]. Model-checking was successfully applied with the tool SIGALI.

An approach that could be compared to ours, although different (uses neither SSA nor synchronous formalisms) is that presented in [HFG08]. The authors define a semantic mapping from SYSTEMC to UPPAAL timed automata in order to get model checking for SYSTEMC designs. It can be observed that our approach, thanks to the different strategies of code generation available for SIGNAL programs [BGT09], also provide simulation code corresponding to the parallel SYSTEMC description, including e.g. static scheduling code.

In this paper, we give in Section 1.2 an overview of the synchronous data-flow language SIGNAL and in Section 1.3 a brief description of SSA, which is the basis for our translation. The principles of the translation from SSA to SIGNAL and its implementation are described in Section 2. Then Section 3 addresses the addition of co-routine parallelism in the programs, using a SYSTEMC flavour. Experimental results are provided for the model-checking of properties on some use case. Some concluding remarks are given in Section 4.

1.2 An overview of SIGNAL

In SIGNAL, a process P consists of the composition of simultaneous equations $x := f(y,z)$ over signals x,y,z. A signal $x \in \mathscr{X}$ is a possibly infinite flow of values $v \in \mathscr{V}$ sampled at a discrete clock noted $\hat{}x$.

$$P,Q ::= x := y\,f\,z \quad | \quad P \texttt{ where } x \quad | \quad P \,|\, Q \qquad \text{(SIGNAL process)}$$

The synchronous composition of processes $P \,|\, Q$ consists of the simultaneous solution of the equations in P and in Q. It is commutative and associative. The process P `where` x restricts the signal x to the lexical scope of P.

An equation $x := y\,f\,z$ denotes a relation between the input signals y and z and an output signal x by a combinator f. An equation is usually a ternary and infixed relation noted $x := y\,f\,z$ but it can in general be an $m+n$-ary relation noted $(x_1,\ldots x_m) := f(y_1,\ldots y_n)$. Such equations are built with usual boolean or arithmetic operations such as `or`, `and`, `not`, $=$, $<$, $+$, $*$, ...

In addition, SIGNAL requires four primitive combinators to perform delay $x := y\$1\,\texttt{init}\,v$, sampling $x := y\ \texttt{when}\ z$, merge $x = y\ \texttt{default}\ z$ and specify scheduling constraints $x \rightarrow y\ \texttt{when}\ \hat{}z$.

The equation $x := y\,f\,z$ where $f = \texttt{or}, =, +, \ldots$ defined the n^{th} value of the signal x by the result of the application of f to the n^{th} values of signals y, z. All signals x, y, z are synchronous (have the same clock): they share the same set of tags $t_1, t_2, \ldots$ (a tag represents a clock tick).

The equation $x := y\$1\,\texttt{init}\,v$ initially defines the signal x by the value v and then by the previous value of the signal y. The signal y and its delayed copy x are synchronous: they share the same set of tags $t_1, t_2, \ldots$ Initially, at t_1, the signal x takes the declared value v and then, at tag t_n, the value of y at tag t_{n-1}.

The equation $x := y\ \texttt{when}\ z$ defines x by y when z is true (and both y and z are present); x is present with the value v_2 at t_2 only if y is present with v_2 at t_2 and if z is present and true at t_2.

The equation $x := y\ \texttt{default}\ z$ defines x by y when y is present and by z otherwise. If y is absent and z present with v_1 at t_1 then x holds (t_1, v_1). If y is present (at t_2 or t_3) then x holds its value whether z is present (at t_2) or not (at t_3).

The equation $x \rightarrow y\ \texttt{when}\ \hat{}z$ forces the constraint that y cannot occur before x when z is present.

In SIGNAL, the presence of a value along a signal x is the proposition noted $\hat{}x$ that is true when x is present and that is absent otherwise. The clock expression $\hat{}x$ can be defined by the boolean operation $x = x$ (i.e. $y := \hat{}x =^{\text{def}} y := (x = x)$). Specific operators are defined on clocks. For instance, $y\,\hat{}+\,z$ is the union of the clocks of signals y, z ($x := y\,\hat{}+\,z =^{\text{def}} x := \hat{}y\ \texttt{default}\ \hat{}z$).

Clock expressions naturally represent control, the clock $\texttt{when}x$ represents the time tags at which the boolean signal x is present and true (i.e. $y := \texttt{when}x =^{\text{def}} y :=\ \texttt{true}\ \texttt{when}\ x$). The clock $\texttt{whennot}\,x$ represents the time tags at which the boolean signal x is present and false. We write $\hat{}0$ for the empty clock (the empty set of tags).

A clock constraint E is a SIGNAL process. The constraint $e\hat{}=e'$ synchronizes the clocks e and e'. It corresponds to the process $(x := (\hat{}e = \hat{}e'))\ \texttt{where}\ x$. Composition $E\,|\,E'$, written also $(|\,E\,|\,E'\,|)$, corresponds to the union of constraints, and restriction $E\ \texttt{where}\ x$ to the existential quantification of E by x.

A useful derived operator is the memory $x := y\ \texttt{cell}\ z\ \texttt{init}\ v$, that allows to memorize in x the latest value carried by y when y is present or when z is true ($x := y\ \texttt{cell}\ z\ \texttt{init}\ v =^{\text{def}} (|\,x := y\ \texttt{default}\ (x\$1\,\texttt{init}\,v)\,|\,x\hat{}=y\,\hat{}+\,(\texttt{when}\,z)\,|)$).

1.3 SSA: an intermediate representation

A program is said to be in static single assignment (SSA) form whenever each variable in the program appears only once on the left hand side of an assignment. Following [CFR+91], a program is converted to SSA form by replacing assignments of a program variable x with assignments to new *versions* $x_1, x_2, \ldots$ of x, uniquely indexing each assignment. Each use of the original variable x in a program block is replaced by the indexed variable x_i when the block is reachable by the i^{th} assignment. For variables in blocks reachable by more than one program block, the ϕ operator is used to choose the new variable value depending on the program control-flow. For example, $x_3 = \phi(x_1, x_2)$ means "x_3 takes the value x_1 when the flow comes from the block where x_1 is defined, and x_2 otherwise". This is needed to represent C programs where a variable can be assigned in both branches of a conditional statement or in the body of a loop.

In this paper, we consider the SSA intermediate representation of the GCC compiler (other modern compilers usually have a similar intermediate format). The compiler provides a language independent, locally optimized intermediate representation for C, C++, and Java programs where programming units such as functions and methods are transformed into a structure in which all native operations are represented by 3-address instructions $x = f(y,z)$. A C program *pgm* is represented by a sequence of labeled blocks *L*:*blk*, where each block is a sequence of statements. Statements may be function calls $x = f(y^*)$ or branches $\texttt{if}\, x\, \texttt{goto}\, L$. Each block is terminated by either a return *rtn* or $\texttt{goto}\, L$ statement. We summarised this representation in Figure 1.

(program)	$pgm ::= L{:}blk \mid pgm;pgm$		(block)	$blk ::= stm;blk \mid rtn$	
(instruction)	$stm ::= x = f(y^*)$	(call)	(return)	$rtn ::= \texttt{goto}\, L$	(goto)
(instruction)	$stm ::= x = \phi(y^*)$	(phi)		$\mid \texttt{return}$	(return)
	$\mid \texttt{if}\, x\, \texttt{goto}\, L$	(test)			

Figure 1: Intermediate representation for C programs

2 Translating sequential code from SSA to SIGNAL

2.1 SSA to SIGNAL: an example

We depict the structure of the SSA form of a typical C program, Figure 2. The function `ones` counts the number of bits set to 1 in a bit-array `data`. It consists of four blocks (labeled `bb_0`, `L0`, `L1`, `L2`). The block labeled `bb_0` initializes the local state variable `idata` to the value of the input signal `data` and `icount` to 0. Then it passes control to the block `L1`. Label `L1` evaluates the termination condition of the loop and passes control accordingly. As there are several possible sources in the control flow for the variables `idata` and `icount`, it determines the most recent value with the help of ϕ functions. If the termination condition is not yet satisfied, control goes to block `L0`, which corresponds to the actual loop contents that shifts `idata` right and adds its right-most bit to `icount`. If the termination condition is satisfied—i.e., all available bits have been counted—control goes to block `L2` where the result is copied to the output signal `ocount`.

```
void ones(int data,
          int *ocount)
{ int icount, idata;
  idata = data;
  icount = 0;
  while (idata) {
     icount += idata & 1;
     idata >>= 1; }
  *ocount = icount;
}
```

```
bb_0: idata_1 = data;
      icount_1 = 0;
      goto L1;
L0:   D = idata_3 & 1;
      icount_2 = D + icount_3;
      idata_2 = idata_3 >> 1;
L1:   idata_3 = PHI<idata_1, idata_2>;
      icount_3 = PHI<icount_1, icount_2>;
      if (idata_3 != 0) goto L0;
L2:   *ocount = icount_3;
      return;
```

Figure 2: From C to static single assignment

Figure 3 depicts the translation of function `ones` into SIGNAL. Signals `data` and `ocount` are respectively input signal (line 2) and output signal (line 3) of the corresponding SIGNAL process.

Lines 4–8 define the labels as boolean signals being true when control flow is in the corresponding block. For instance, `bb_0` is true at the first instant, then it is false forever; `L1` is true when either `L0` or `bb_0` is (control can go to `L1` from `L0` or `bb_0`). Note that there is no need to introduce a delay when control passes to `L1`. This is not the case for `L0`, for which there is a transition from `L1` when the termination condition of the loop is not satisfied ((`idata_3`$/=0$) `when L1`): in that case, control will be in `L0` at the *next step*.

Lines 14–15, 17–19 and 24 represent respectively the computations that are done in blocks `bb_0`, `L0` and `L2`: this is expressed by the sampling (`when`) on the corresponding boolean. Note that state variables are necessary to memorize the values of `idata_3` and `icount_3` (lines 10–11). Line 18, the operands of the plus operator have to be memorized at some common clock (`_pk_1`, line 27) since the arguments of the plus must be synchronous. The ϕ functions of block `L1` are simply defined with merge (`default`) operators in SIGNAL (lines 21–22).

```
1  process ones =
2    ( ? integer data;
3      ! integer ocount; )
4  (| (| bb_0 := (not (^bb_0)) $1 init true
5      | next_L0 := ((idata_3/=0) when L1) default false
6      | L0 := next_L0 $1 init false
7      | L1 := (true when L0) default (true when bb_0)
8      | L2 := (not (idata_3/=0)) when L1
9      |)
10   | (| Z_idata_3 := idata_3 $1
11     | Z_icount_3 := icount_3 $1
12     |)
13   | (| data_1 := data cell (^bb_0) |)
14   | (| idata_1 := data_1 when bb_0
15     | icount_1 := 0 when bb_0
16     |)
17   | (| D := bit_and(Z_idata_3, 1) when L0
18     | icount_2 := ((D cell _pK_1)+(Z_icount_3 cell _pK_1)) when L0
19     | idata_2 := bit_right_shift(Z_idata_3, 1) when L0
20     |)
21   | (| idata_3 := idata_2 default (idata_1 default Z_idata_3)
22     | icount_3 := icount_2 default (icount_1 default Z_icount_3)
23     |)
24   | ocount_1 := icount_3 when L2
25   | (| when bb_0 ^= data
26     | bb_0 ^= L0 ^= idata_3 ^= icount_3 ^= data_1
27     | _pK_1 := Z_icount_3 ^+ D
28     |)
29   | (| ocount := (ocount_1 cell L2) when L2 |)
30   |)
31 where ... end;
```

Figure 3: From SSA to SIGNAL

2.2 SSA to SIGNAL: translation scheme

A general scheme for the translation is described in Figure 4 with a function $\mathscr{I}[\![pgm]\!]$, defined by induction on the formal syntax of *pgm*. The overall idea is to translate one (or several) SSA basic blocks into a parallel assignment in SIGNAL. The sequence of instructions in SSA is then considered within a SIGNAL clock tick. In the presence of loops (i.e. backward gotos), we must represent the successive values taken by variables at different laps of the loop with different clock ticks. The control flow is modeled with the notion of clock: a parallel assignment in SIGNAL has its clock activated when the corresponding piece of SSA code would be executed.

The present value of a signal is noted x, its next value is noted x'. With each block of label $L \in \mathscr{L}_f$ in a given function f, the function $\mathscr{I}[\![pgm]\!]$ associates an *input clock* x_L, an *immediate clock* x_L^{imm} and an *output clock* x_L^{exit} (note that all these clocks are not necessarily generated in the effective translation). The clock x_L is true iff L has been activated in the previous transition (by emitting the event x'_L). The clock x_L^{imm} is set to true to activate the block L immediately. The clock x_L^{exit} is set to true when the execution of the block labeled L terminates. The default activation condition of this block is the clock $x_L \vee x_L^{imm}$ (union of clocks x_L and x_L^{imm}: equation (1) of Figure 4). The block *blk* is executed iff the proposition $x_L \vee x_L^{imm}$ holds, meaning that the program counter is at L.

For a return instruction or for a block, the function returns a SIGNAL process P. For a block instruction *stm*, the function $\mathscr{I}[\![stm]\!]_L^{e_1} = \langle P \rangle^{e_2}$ takes three arguments: an instruction *stm*, the label L of the block it belongs to, and an input clock e_1. It returns the process P corresponding to the instruction and its output clock e_2. The output clock of *stm* corresponds to the input clock of the instruction that immediately follows it in the execution sequence of the block.

Rules (1-2) in Figure 4 are concerned with the iterative decomposition of a program *pgm* into blocks *blk* and with the decomposition of a block into *stm* and *rtn* instructions. In rule (2), the input clock e of the block *stm*;*blk* is passed to *stm*. The output clock e_1 of *stm* becomes the input clock of *blk*.

$$
\begin{array}{lll}
(1) & \mathscr{I}[\![L{:}blk;pgm]\!] = \mathscr{I}[\![blk]\!]_L^{x_L \vee x_L^{imm}} \mid \mathscr{I}[\![pgm]\!] \\
(2) & \mathscr{I}[\![stm;blk]\!]_L^e = \text{let } \langle P \rangle^{e_1} = \mathscr{I}[\![stm]\!]_L^e \text{ in } P \mid \mathscr{I}[\![blk]\!]_L^{e_1} \\
(3) & \mathscr{I}[\![\texttt{if}\, x\, \texttt{goto}\, L_1]\!]_L^e = \langle \mathscr{G}_L(L_1, e \wedge x) \rangle^{e \wedge \neg x} \\
(4) & \mathscr{I}[\![x = f(y^*)]\!]_L^e = \langle \mathscr{E}(f)(xy^*e) \rangle^e \\
(5) & \mathscr{I}[\![\texttt{goto}\, L_1]\!]_L^e = (e \Rightarrow x_L^{exit} \mid \mathscr{G}_L(L_1, e)) \\
(6) & \mathscr{I}[\![\texttt{return}]\!]_L^e = (e \Rightarrow (x_L^{exit} \mid x_f^{exit})) \\
\text{where} & \mathscr{G}_L(L_1, e) = \text{if } L_1 \textit{ is after } L \textit{ in the control-flow} \text{ then } e \Rightarrow x_{L_1}^{imm} \text{ else } e \Rightarrow x'_{L_1} \\
& \mathscr{E}(f)(xyze) = e \Rightarrow (\hat{x} \mid x = [\![f]\!](y,z)), \forall f x y z e
\end{array}
$$

Figure 4: Translation scheme

The input and output clocks of an instruction may differ. This is the case, rule (3), for an $\texttt{if}\, x\, \texttt{goto}\, L_1$ instruction in a block L. Let e be the input clock of the instruction. When x is false,

then control is passed to the rest of the block, at the output clock $e \wedge \neg x$ (intersection of clocks e and $\neg x$). Otherwise, the control is passed to the block L_1, at the clock $e \wedge x$.

There are two ways of passing the control from L to L_1 at a given clock e. They are defined by the function $\mathscr{G}_L(L_1, e)$: either immediately, by activating the immediate clock $x^{imm}_{L_1}$, i.e., $e \Rightarrow x^{imm}_{L_1}$ (the notation $e \Rightarrow P$ means: if e is present then P holds); or by a delayed transition to L_1 at e, i.e., $e \Rightarrow x'_{L_1}$. This choice depends on whether L_1 is after L in the control flow, i.e. whether the block L_1 can be executed immediately after the block L.

Rule (4) is concerned with the translation of native and external function calls $x = f(y^*)$. The generic translation of f is taken from an environment $\mathscr{E}(f)$. It is given the name of the result x, of the actual parameters y^* and of the input clock e to obtain the translation of $x = f(y^*)$. This translation works when there is only one call to f at the same time. Recursive calls of f would require an explicit stack, and parallel calls would require duplicating the generated code for f for each thread. The generic translation of 3-address instructions $x = f(y,z)$ at clock e is given by $\mathscr{E}(f)(xyze)$.

Instructions `goto` and `return`, rules (5-6), define the output clock x^{exit}_L of the current block L by their input clock e. This is the right place to do that: e defines the very condition upon which the block actually reaches its return statement. A `goto`L_1 instruction, rule (5), passes control to block L_1 unconditionally at the input clock e by $\mathscr{G}_L(L_1, e)$. A `return` instruction, rule (6), sets the exit clock x_f to true at clock e to inform the caller that f is terminated.

2.3 C/C++ to SIGNAL: implementation

SIGNAL models are automatically generated from C/C++ component descriptions with the help of the GNU Compiler Collection (GCC) [Fre] and its static single assignment (SSA) intermediate representation [GCC03, The]. This is obtained in three main stages, as described below:

1. **Converting C/C++ into SSA:** The first step of the translation scheme consists in converting C/C++ models into the SSA form. This step is performed by GCC, which goes through several intermediate formats (Gimple Trees, then Control-Flow Graph (CFG)), and then produces the SSA form which is used by GCC for optimizations.
2. **Converting SSA into SIGNAL:** The next step of the translation scheme consists in converting SSA into SIGNAL processes. It is implemented in the GCC front-end. The output of this step is a SIGNAL program which reflects directly the SSA code in a SIGNAL syntax (but without a correct semantics at this point). The implementation of the SIGNAL generation is inserted in the GCC source tree as an *additional front-end optimization pass*. GCC currently features over fifty optimization passes. It can be chosen to use all of these by inserting this additional pass at the very end, but it may also make sense to exclude some of the optimizations. The resulting syntactic SIGNAL program is another view of the SSA code without any transformation (so the connexion to some other C compiler with an SSA internal representation would be easily possible). This code is composed of a set of labeled blocks, composed of a set of ϕ definitions, a set of computations, and a branching.
3. **Transforming the SIGNAL program:** The next step consists in the definition of (i) the control induced by the references to the labels in the branching statements; (ii) the memories induced by the loops and the control. The control is given first to the first block (through the signal `bb_0` in the function `ones` example).

3 Modeling parallelism in the SSA to SIGNAL line

The previous section described a translation of sequential, imperative code, into SIGNAL. We now present a way to extend this translation scheme to parallel code. We consider the case of co-routine semantics (i.e. non-preemptive scheduling with a single processor).

SYSTEMC is an example of an execution platform with co-routine semantics. It is built on top of the C++ language, augmented with a scheduler, and communication and synchronization primitives. We implemented a translation from a small subset of SYSTEMC which has basically two elements with respect to parallelism: pieces of code to be executed "atomically", and a `yield()` instruction, that stops the execution of a thread, and yields the control back to the scheduler. The scheduler then elects any thread, non-deterministically.

The official SYSTEMC library doesn't have a `yield()` instruction, but this instruction can be implemented either with a slight modification of the scheduler as proposed by Claude Helmstetter [Hel07], or more simply by a `wait(random(), SC_NS);`. The motivation for choosing `yield()` instead of the usual `wait()` statements of SYSTEMC is to start with the simplest scheduling, to keep the focus on the notion of parallelism. We will show latter that implementing an arbitrary scheduling policy on top of this is possible.

In this subset of SYSTEMC, we do not have any specific communication and synchronization primitives, but processes can communicate using shared variables.

3.1 Presentation of SYSTEMC

The core of the SYSTEMC syntax relevant to the present study is represented in Figure 5. A system consists of the composition of classes and modules *sys*. A class declaration $\mathsf{class}\, m\,\{dec\}$ associates a class name m with a sequence of fields *dec*. It is optionally parameterized by a class with $\mathsf{template}\,\langle \mathsf{class}\, m_1\rangle$. To enforce a strong typing policy, we annotate the class parameter m_1 with $\#\mathsf{TYPE}(m_1, m_2)$ to denote the type of m_1 with the virtual class m_2. A module $\mathsf{SC\_MODULE}(m)$ is a class that defines an architecture component. Its constructor $\mathsf{SC\_CTOR}(m)\ \{new; pgm\}$ allocates threads (e.g. $\mathsf{SC\_THREAD}(f)$) and executes an initialization program *pgm*. Modules define threads whose execution is concurrent. Declarations *dec* associate locations x with native classes or template class instances $m\langle m^*\rangle$, and procedures with a name f and a definition *pgm*. *pgm* can be any C++ code. We assume x to denote the name of a variable or signal and to be possibly prefixed as $m :: x$ by the name of the class it belongs to.

$$
\begin{array}{lll}
sys & ::= [\,\mathsf{template}\,\langle \mathsf{class}\, m_1\rangle\ \#\mathsf{TYPE}(m_1, m_2)\,] \quad \mathsf{class}\, m\,\{dec\} & \text{(class)} \\
 & \mid \mathsf{SC\_MODULE}(m)\ \{dec; \mathsf{SC\_CTOR}(m)\ \{new\}\}; & \text{(module)} \\
 & \mid sys \quad sys & \text{(sequence)} \\
dec & ::= m\langle m^*\rangle\, x & \text{(field)} \\
 & \mid \mathsf{void}\, f()\ \{pgm\}; & \text{(thread)} \\
 & \mid dec \quad dec & \text{(sequence)} \\
new & ::= \mathsf{SC\_THREAD}(f); \quad \mathsf{sensitive} \ll x^* \quad \mid \quad new; pgm & \text{(constructor)}
\end{array}
$$

Figure 5: Abstract syntax for SYSTEMC

A simple example in given in Figure 6. It defines two n-bits counters in parallel. The macro `DECLARE_COUNTER` declares n boolean state-variables b_i, the function `step` performs one step (applying next$(b_i) = b_i$ xor c_{i-1} and $c_i = b_i$ and c_{i-1}, c_i being the carry), and the macro `BEGINNING_OF_PROCESS` declares the local variables c_i. Each counter comes with two additional variables `..._started` and `..._finished`, maintained up to date by `step()`, that are true respectively after the counter did its first step, and once each bit of the counter is true.

```
SC_MODULE(module)
{
   DECLARE_COUNTER(count1_);
   DECLARE_COUNTER(count2_);
   void compute1() {
      BEGINNING_OF_PROCESS;
      while(!(count2_started))
        { yield(); }
      while(!(count1_finished))
        { step(count1_);
          yield(); }
      ASSERT(count2_finished);
   }
   void compute2() {
      BEGINNING_OF_PROCESS;
      while(!(count2_finished))
         { step(count2_);
           // yield();
         }
   }
   SC_CTOR(module) {
        INIT_COUNTER(count1_);
        INIT_COUNTER(count2_);
        SC_THREAD(compute1);
        SC_THREAD(compute2);
   }
};
```

Figure 6: Parallel counters

3.2 Principle of the translation for SystemC code

In SystemC, the bodies of processes are described using plain C++ code (plus function calls to the SystemC API, i.e. `yield` in our example). As a consequence, the translation from C/C++ to Signal can be reused for that purpose with a few adjustments, detailed in the following.

3.2.1 Isolation of system calls

First it can be noticed that GCC considers SystemC macros (including synchronization primitives ("system calls") like `yield` as plain C++. As opposed to that, our approach requires a special handling of these macros in the Signal code. Thus they have first to be visible in the Ssa code generated for the SystemC threads. To this end, they have to be viewed by GCC as external function calls. This is the case, for instance, for the instruction `yield` used in the program of Figure 6: it is passed as such in the Ssa code.

However, if system calls are processed by GCC as usual external calls, they are not distinguished from other instructions in the Ssa code and they may appear among other computations in Ssa labeled blocks. A first requirement for being able to process system calls specifically in Signal is thus to isolate them in specific blocks in Ssa. This is an easy transformation that consists in breaking up the blocks containing system calls, while respecting the control flow. In the Ssa to Signal transformation, it is implemented as the very first step of the transformations applied on the syntactic Signal code (see Section 2.3, step 3). In the resulting Signal translation, the label of the block containing a system call will be viewed as the activation clock of the

call of the primitive. Then, suppose that `l0`, ..., `ln` are the labels corresponding to the different calls of a given primitive (say `yield`, for instance) in a given thread, then the following signal: `_yield_` := (`when l0`) `default ... default` (`when ln`) represents the clock at which control has to be returned to the scheduler, from a `yield` primitive, for the considered thread. Also, it is necessary to take into account that the block that follows a system call cannot be run in the same logical instant than this system call (the OS has to take the control). Thus, the signal representing the label of this block has to be delayed by one instant and additional memories may be required for some variables.

3.2.2 Addition of control signals

In the C to SIGNAL translation, input and output signals of the SIGNAL process resulting from the translation of a C procedure correspond to the parameters of the procedure. When translating a thread in a multi-thread context, a few input or output control signals have to be added, in order to communicate with the system. These signals are the following:

input signal `running`: This signal is defined by the system. It specifies the clock at which the process is actually running (the processor is attributed to this process). Remind that in the SIGNAL code obtained from a SSA form, each operation is sampled, either directly or recursively, by the clock corresponding to a given label (for instance, `ocount_1 := icount_3 when L2`). In the process corresponding to a thread, each label is additionally sampled by the signal `running` (for instance, `L2 := (not (idata_3/=0)) when L1 when running`).

output signal `end_processing`: This signal is defined by the clock corresponding to the final block of the SSA (for the example of Figure 3, it would be: `end_processing := when L2`). This is the way for processes to inform the scheduler that a yielding instruction was reached, letting the scheduler decide which process to wake up after.

output signals corresponding to system calls in the thread: for example, a signal `_yield_`, as defined above, is produced, corresponding to the clock of the calls of the `yield` primitive in the thread. Other signals correspond to `wait` or `signal` primitives, for instance. If the primitives are not used in the process, their clock is the empty clock. These signals complement `end_processing` in that `end_processing` says *whether* a yielding instruction was reached, while other signals like `_yield_` tell *which* one.

These signals are added automatically in the translation when the application is a multi-threaded one. Note that another input signal, `start`, can be added if restart is possible. It is then used to replace the definition of the initial label of the process: `bb_0 := (start default false) when running`.

3.2.3 Shared variables

Care has to be taken for variables shared by different threads. First, when a variable is not declared in the procedure where it is used, GCC does not produce real SSA for these variables: there is no creation of distinct instances for them, no ϕ function for merging their different definitions. They are considered as "virtual SSA". For these variables, the mechanism of *partial definitions* provided in SIGNAL is used. Let `x` be such a variable and suppose one of its definitions is $\mathtt{x} = E$

in a SSA block labeled `li`. The corresponding SIGNAL definition will be: `x ::= E when li` (the use of `::=` means that there are possibly other definitions for `x`). The shared variables are necessarily state variables in SIGNAL, for which their previous value is kept, when they are not redefined.

3.2.4 Inclusion in a simulator

In order to validate the translation scheme described above, a mock-up scheduler is described in SIGNAL. This scheduler contains a non-preemptive scheduler that attributes non-deterministically the processor to one process when several processes are ready to execute. This corresponds to the SYSTEMC scheduler with the `yield()` instruction described above. In SIGNAL, if `conflict` represents the clock at which there is a potential conflict for the attribution of the processor (several processes are ready), the running process is chosen by: `pid := (any when conflict) default ...`, where `any` is an input signal that can take any value. The scheduler manages the status of each one of the processes p_i corresponding to the threads of the application. A SIGNAL process `SET_STATUS` is associated with each p_i, with state variables representing the current and next status of p_i (*ready, running*). The scheduler receives and uses the control signals that are produced in the processes p_i. For instance, the clock of the signal `_yield_` produced in some process p_k defines instants at which the next status of p_k, whose current status is *running*, will be *ready* (so that the scheduler will have to choose, non-deterministically, a new running process). Thus, in return, the scheduler defines the control signals `running` provided to each one of the p_i's. For a given p_k, the corresponding signal `running` represents the clock at which p_k has the *running* status.

It is worth noticing that the control of a given application is represented very differently in SIGNAL than it would be in some usual imperative parallel language. There is no explicit program counter, no imperative description of *suspend* or *resume* actions. The control is fully described by the clocks of the signals of the application. The SIGNAL equations defining some process p_i define its behavior as invariant equations. We explained that all operations in p_i are conditioned by a given input signal `running`. Periods of time in which p_i is not running (or is otherwise suspended) correspond to the instants at which the signal `running` is absent. So that suspend/resume, for instance, is automatically handled through the clock of the signal `running`.

3.2.5 Possible extensions

The scheduler described above is very simple. With the signals `running`, and `end_processing`, it can model a non-preemptive scheduler. By adding more signals between the scheduler and the processes, and a more complex state-machine in the scheduler, one can relatively easily model a more complex scheduler, like the complete scheduler of SYSTEMC. Indeed, a similar approach was followed in the tool LusSy [MMM06] and could easily be adapted, since LusSy also uses a synchronous formalism to model the scheduler. The scheduler used in LusSy omits a few details of the actual scheduler specifications, but a more complete version is described in [TKVS08], and even this version is still only a dozen states, and could be modeled with a few tens of lines of code in SIGNAL.

The main difference with the translation implemented in LusSy is that the later does not use

SSA as an intermediate form, and is indeed less efficient for the translation of plain C++ code (a more detailed comparison of the tools is in progress and will not be detailed here).

3.3 Experiments

The example described in Section 3.1 (Figure 6) is used for basic experiments. The program is automatically translated into SIGNAL via SSA following the general scheme described above. Then simulation code (in C) is generated from the SIGNAL program with the POLYCHRONY toolset [INRa]. Traces have been added in the SIGNAL program to be able to follow the simulation. The results are those expected, whatever is the choice of the scheduler.

Besides these first results, a main objective of the experiments is to demonstrate the possibility of formal validation of models using this methodology. We use again the same example (*parallel counters*) to prove formal properties using model-checking. The first counter waits that the second one has started counting to count also. At the end, it checks that the second counter has finished (property `ASSERT(count2_finished)`). Indeed, from the point of view of the first counter, when the second one has started, it has also terminated, so that the property is true. A variant of the program (*parallel counters with variant*) is when a `yield()` is introduced in the body of the loop of the second counter. In that case, it is possible to start the second counter without finishing it, and then the first counter can run till the end, so that the property is false.

The SIGNAL compiler included in the POLYCHRONY toolset allows for checking *static* properties such as contradictory clock constraints, cycles, null clocks, exclusive clocks... In order to check *dynamic* properties, the SIGNAL companion model-checker SIGALI [INRb, MR] may be used. It is an interactive tool specialized on algebraic reasoning in $\mathbb{Z}/3\mathbb{Z}$ logic. SIGALI transforms SIGNAL programs into sets of dynamic polynomial equations that basically describe an automaton. Then it can analyze this automaton and prove properties such as liveness, reachability, and deadlock. The fact that it is reasoning only on a $\mathbb{Z}/3\mathbb{Z}$ logic constrains the conditions to the boolean data type (true, false, absent). This is practical in the sense that true numerical verification very soon would result in state spaces that are no longer manageable, however it requires, depending on the nature of the underlying model, major or minor modifications prior to formal verification. For many properties, numerical values are not needed at all and can be abstracted away thus speeding up verification. When verification of numerical manipulations is sought, an abstraction to boolean values can be performed (like replacing any condition depending on integers with non-deterministic Boolean), that suffices in most cases to satisfy the needs. Note that the translation to the SIGALI format is done after the so-called *clock calculus* completed by the SIGNAL compiler. This clock synthesis allows to reduce significantly the number of constraints. Unfortunately, SIGALI does not provide counter-examples for the system when the proof fails.

For the *parallel counters* example and its variant, all data types are boolean (and the mock-up scheduler has been encoded also using only boolean types). The results of the verification of properties using SIGNAL and SIGALI are those expected. Performances (time and size of the system) for obtaining the results are provided in Figure 7 for 2-bits and 8-bits versions of the counters (they are obtained using a reordering of variables in SIGALI).

program	size				time
	state var.	states	reach. states	transitions	
2-bits parallel counters (property true)	24	2^{24}	36	116	0.15 s
2-bits parallel counters with variant (prop. false)	25	2^{25}	107	359	0.27 s
8-bits parallel counters (property true)	36	2^{36}	1.296	3.896	66 s
8-bits parallel counters with variant (prop. false)	37	2^{37}	328.715	1.117.223	124 s

Figure 7: Performances for proving properties with SIGALI

3.4 Discussions on performances

One common mis-conception about SSA is that since multiple assignments to the same variable are translated into assignments to multiple intermediate variables, the explosion of the number of variables introduces a huge overhead. This paper shows that the explosion of the number of variables is indeed not a problem: most variables are encoded into temporary variables in SIGNAL, and will be dealt with very efficiently by a model-checker (no additional nodes in BDDs). Our experiments show that the number of *state* variables does not explode.

On the other hand, one real advantage of this approach is that it creates very few transitions in the generated code. In the absence of loops, a portion of code between two yielding instructions is indeed encoded in one clock tick in the synchronous world. As opposed to this, a naive approach translating each instruction with an explicit control-point would generate huge automata, with a lot of control-points. The encoding of this automaton would introduce either a lot of Boolean state variables (with a one-hot encoding) or state variables with a large number of possible values. Our SSA-based translation avoids this overhead.

4 Conclusion

We described the principle and implementation of a translation of imperative code into the synchronous data-flow language SIGNAL. Using SSA as an intermediate format, the translation is natural. Since the SSA form we are using (the one of GCC) is very simple, the implementation does not have to deal with all the particular cases of the source language (C++), and can focus on optimizations and performances. We also showed that the extension to a simple model of parallelism was possible and relatively straightforward, and showed the way to encode a more complex scheduling policy.

The main limitation of the approach with respect to parallelism is that although the co-routine semantics (non-preemptive scheduling) is encoded as a natural extension of the sequential translation, a preemptive scheduling, or even real parallelism, would be much harder to model faithfully with this principle.

Indeed, the main point of our approach is to encode a sequence of assignments, in one or several basic-blocks, into a single parallel assignment, within one clock tick. This turns the sequential piece of code into large atomic actions, which were not atomic in the original code. In

other words, applying the translation naively would reduce the set of possible interleaving, thus reducing the set of possible behaviors, and missing bugs in the proof.

Applying the translation to real parallel code would therefore require identifying which portions of code can be considered atomic, and where to introduce preemption points. Each preemption point could then be encoded in the same way as the `yield()` instruction. Actually, identifying which section can be considered atomic, which instructions can permute and move from a section to another, is an active research domain (see for example [QRR04]).

Another limitation of the current implementation is that we currently manage only a small subset of SystemC. Modeling the scheduling algorithm itself would probably not be the most difficult part. One bigger difficulty is to take the architecture of the platform into account. For example, when one process writes `wait(event);` and the other writes `event.notify();`, the translation obviously has to take into account the fact that `event` is the same object in both cases. Another example is when one does a `port.write(value);` and the other a `another_port.read();` in this case, the translation has to depend on whether `port` and `another_port` are bound together or not. Extracting such information from SystemC code requires a SystemC front-end, and not just a C++ one. Many such front-ends exist, like Pinapa [MMM05] used by LusSy, but none of them use Ssa as an intermediate representation.

Unfortunately, re-using an existing compiler to get both an Ssa intermediate form and the architecture of the platform, linked together, is not easy [BGM+08]. The approach followed by Pinapa does reuse an existing compiler, but relies partly on the fact that the intermediate format is a high-level abstract syntax tree. We are working on a new version of Pinapa that would use an Ssa-based compiler, but this requires a substantial rework of the approach, and a complete rewrite of the code itself.

Bibliography

[BB91] A. Benveniste, G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE* 79(9):1270–1282, Sep. 1991.

[BCE+03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone. The synchronous languages 12 years later. In *Proceedings of The IEEE*. Pp. 64–83. 2003.

[BGM+08] L. Besnard, T. Gautier, F. Maraninchi, M. Moy, J.-P. Talpin. Comparative study of approaches to semantics extraction and virtual prototyping of system-level models. Technical report, Verimag, Grenoble INP, France; IRISA, INRIA Rennes, France, 2008. http://www-verimag.imag.fr/~moy/fotovp/rapport-fotovp.pdf.

[BGT09] L. Besnard, T. Gautier, J.-P. Talpin. Code generation strategies in the Polychrony environment. Research report RR-6894, INRIA, 2009. http://hal.inria.fr/inria-00372412/en/

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(4):451–490, 1991.

[CGP99] E. Clarke, O. Grumberg, D. Peled. *Model checking*. Springer, 1999.

[Fre] Free Software Foundation. The GNU Compiler Collection. http://gcc.gnu.org.

[GCC03] *Proceedings of the 2003 GCC Developers Summit*. Ottawa, Ontario Canada, 2003.

[Hel07] C. Helmstetter. *Validation de modèles de systèmes sur puce en présence d'ordonnancements indéterministes et de temps imprécis*. PhD thesis, INPG, Grenoble, France, March 2007.
http://www-verimag.imag.fr/~helmstet/these-fr.html

[HFG08] P. Herber, J. Fellmuth, S. Glesner. Model checking SystemC designs using timed automata. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. Pp. 131–136. ACM, New York, NY, USA, 2008.

[INRa] INRIA Espresso Team. Polychrony tool. http://www.irisa.fr/espresso/Polychrony.

[INRb] INRIA Vertecs/Espresso Teams. Sigali tool. http://www.irisa.fr/vertecs/Softwares/sigali.html.

[KTBB06] H. Kalla, J.-P. Talpin, D. Berner, L. Besnard. Automated translation of C/C++ models into a synchronous formalism. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*. Pp. 426–436. March 2006.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE* 79(9):1321–1336, Sep. 1991.

[LTL03] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers* 12(3):261–304, April 2003.

[MMM05] M. Moy, F. Maraninchi, L. Maillet-Contoz. Pinapa: An Extraction Tool for SystemC descriptions of Systems-on-a-Chip. In *EMSOFT*. Pp. 317 – 324. September 2005.

[MMM06] M. Moy, F. Maraninchi, L. Maillet-Contoz. LusSy: an open Tool for the Analysis of Systems-on-a-Chip at the Transaction Level. *Design Automation for Embedded Systems*, 2006. special issue on SystemC-based systems.
http://www-verimag.imag.fr/~moy/publications/springer.pdf

[MR] H. Marchand, E. Rutten. Sigali User Manual. http://www.irisa.fr/espresso/Polychrony.

[QRR04] S. Qadeer, S. K. Rajamani, J. Rehof. Summarizing procedures in concurrent programs. In *POPL '04: Proceedings of the 31st symposium on Principles of programming languages*. Pp. 245–255. ACM, New York, NY, USA, 2004.

[The] The Tree SSA project. Tree-SSA. http://gcc.gnu.org/projects/tree-ssa.

[TKVS08] D. Tabakov, G. Kamhi, M. Vardi, E. Singerman. A Temporal Language for SystemC. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08.* Pp. 1–9. 2008.

[TLSG05] J.-P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta. A compositional behavioral modeling framework for embedded system design and conformance checking. *International Journal of Parallel Programming* 33(6):613–643, 2005.

Automatically Generating CSP Models for Communicating Haskell Processes

Neil C. C. Brown

Computing Laboratory, University of Kent, UK

Abstract: Tools such as FDR can check whether a CSP model of an implementation is a refinement of a given CSP specification. We present a technique for generating such CSP models of Haskell implementations that use the Communicating Haskell Processes library. Our technique avoids the need for a detailed semantics of the Haskell language, and requires only minimal program annotation. The generated CSP_M model can be checked for deadlock or refinements by FDR, allowing easy use of formal methods without the need to maintain a model of the program implementation alongside the program itself.

Keywords: CSP, Automatic Model Generation, Haskell

1 Introduction

Programs designed using formal methods such as Communicating Sequential Processes (CSP) [Hoa85, Ros97] typically have a specification and a more complicated implementation. A tool such as FDR [For97] can be used to check that the implementation is a refinement of the specification. However, determining the CSP model of an implementation – written in an executable programming language using a CSP-based library – can be difficult due to problems with unclear semantics, or translation errors. Generating a model from the program should thus be done automatically.

Generating a formal model of an existing program typically requires complete access to the program's source code (including libraries) and a detailed semantics of the programming language. Both requirements can be problematic: libraries may be closed source, and the programming language may lack a detailed semantics, especially a platform independent semantics that includes the semantics of the threading system and memory model.

Communicating Haskell Processes (CHP) [Bro08] is a library for the functional programming language Haskell with a strong correspondence to CSP. CHP allows for execution of CSP-like programs, combining the concurrency concepts of CSP with the expressive power of Haskell.

In this paper we describe a technique for generating formal models of CHP programs without the need for source code analysis. We take advantage of Haskell's purity and thus do not require a semantics of the Haskell language. We describe the class of programs that can be modelled, along with several examples. Our technique can be used to generate models to check for refinement of specifications, or for generating models of rapidly prototyped programs (for example, using agile development methods).

2 CHP

Although Haskell is a functional programming language, it has support for imperative programming through the concept of monads: a monad captures a common pattern that can be used for imperative programming. Thus, a CHP program can be conceptually separated into the pure functional computations, and the imperative communication aspects.

For example, this is a map process in CHP that transforms items as they pass through:

```
map :: (a -> b) -> Chanin a -> Chanout b -> CHP ()
map f input output = do x <- readChannel input
                        writeChannel output (f x)
                        map f input output
```

The top line is the type signature of the process. Types that begin with a lower-case letter are parameterised types – thus this type reads: given a function from some type a to some type b, an incoming channel of type a and an outgoing channel of type b, the process will yield a CHP process with the `()` unit type as its return.

The implementation of the process is straightforward. A value is read from the input channel and bound to the name x. The result of applying the function f to x is sent on the output channel, and then the process recurses. It can be seen that the behaviour of the pure function f has no effect on the communication behaviour of the process (providing that f terminates).

3 Approach

Our approach is to take the CHP library and provide a mirror implementation with near-identical Application Programming Interface (API). This mirror implementation does not properly execute the code as the original CHP would, but instead traces[1] the structure of the program and produces a CSP model of the program.

The original library has API functions for creating channels and barriers, and communicating and synchronising on them (see section 7.1 for details on the synchronisation semantics). The mirror version has the same creation functions, but the internal definition of channels and barriers changes so that they simply hold a unique identifier. The communication and synchronisation functions are changed in the mirrored version to record the synchronisation in a CSP model, rather than performing an actual synchronisation.

The mirror definitions for the parallel composition operator and the external choice operator generate the models for all processes involved, and then compose the models using parallel composition and external choice. The monad system in Haskell means that we can alter the definition of sequential composition (akin to overloading the semi-colon in C++ or Java) to compose the models in sequence. Thus only the pure computations might be executed[2], but none of the imperative side-effecting parts of the process. This is because the latter are replaced by code that only generates the model. The three types of imperative statements that can occur in the CHP monad are:

[1] The sense of tracing used here is not related to the idea of traces in CSP.

[2] Because of Haskell's laziness, these will only be executed if the result is required to make a choice about the program's flow (see section 3.1 for more details).

1. Channel/barrier creation,
2. Channel communication and barrier synchronisation, and
3. Lifted IO actions (e.g. writing to a file, opening a network socket).

We have already explained the first two; IO actions are described in section 5. Given the existence of this mirror library, the main change needed to generate a model rather than run a program is to change the normal *import Control.Concurrent.CHP* – perhaps using a pre-processor for ease of use – to *import Control.Concurrent.CHP.Model*.

This library substitution approach avoids the need for source code analysis, which means that no source code is used by the tool. One drawback to this is that looping and recursive behaviours cannot be detected automatically, and require extra annotations (see section 4).

3.1 Reading from Channels

Our approach benefits from Haskell's pure and lazy nature. Many processes, such as the earlier *map* process, have behaviour that is unaffected by the values being transmitted. It is thus possible to return the value $\perp$ from reading a channel, i.e. a value that will give an error if evaluated. The consequences of this change are as follows.

- If the value read from the channel is discarded, returning $\perp$ will not have any adverse effect.
- If the value is sent on another channel, this will also not cause an error as we re-implement the channel output function. We are able to track the different $\perp$ values throughout the system to know which channel the value came from.
- If the value is used as part of another computation that does not end up being evaluated (e.g. because the new value is sent down a channel, or is not needed), there will again not be a problem because of Haskell's laziness.
- If the value is evaluated because it is used to determine the program's flow, this will cause the error from the $\perp$ value to occur. This is explored further in section 5.

3.2 Value Sources

The separation of monadic side effects from pure code means that any value in a CHP process must come from one (or a combination) of three sources:

1. A deterministic source, e.g. a constant in the program, or a pure computation with constant inputs.
2. A value read from a communication channel, or returned from an IO action.
3. A parameter to the process.

Values in the first category are the same every time the program is run, and thus any decisions about the program's behaviour based solely on these values will always have the same outcome. Values in the second category are dealt with in section 5, while values in the third category are explained in section 4.

It should be emphasised that our approach is not the same as producing a model based on observation of executing the process. The combination of the replacement of the monad, and the use of $\perp$ bottom values means that we can ensure that the model we generate is an accurate reflection of the process's behaviour in *every* execution, with the exception of issues dealt with in later sections of this paper, and known Haskell "back doors" such as the aptly named *unsafePerformIO*.

4 Recursion and Looping

Most processes have infinite behaviour, via the use of recursion. A very straight-forward example is a process that consumes input:

```
blackHole :: Chanin a -> CHP ()
blackHole input = do readChannel input
                     blackHole input
```

This can also be written using the common Haskell function *forever* that infinitely repeats in sequence a monadic action:

```
blackHole' :: Chanin a -> CHP ()
blackHole' input = forever (readChannel input)
```

Infinite behaviour cannot be detected with our substituted monad; there is no way to determine after one million consecutive inputs whether the next statement will definitely be a input. We must require the programmer to add an annotation to aid in spotting recursion.

The *process* annotation is placed as follows:

```
blackHole :: Chanin a -> CHP ()
blackHole = process "blackHole" inner
  where
    inner input = do readChannel input
                     blackHole input
```

The *process* annotation uses Haskell's type-classes to take as its second argument a process with N arguments, and return a process that takes the same N arguments. The behaviour of the *process* wrapper function is to record the arguments to the process ready for future equality checking.

When *blackHole* is first called with a channel c, the *process* annotation stores a pair of the process name and a list containing all the arguments (in this case, just c) as the key in an associative map with a placeholder for the value. When the process then recurses, the process name and list of arguments (which is again, the singleton list with c) is looked up in the map. A value is found, which causes the execution of the process to return the placeholder. Thus only one execution of the process is examined, and the recursive call is skipped.

The analysis of the *blackHole* process is then complete, and the placeholder is substituted for a definition of the process's behaviour. This relies on the user-provided guarantee that all parameters that affect a process's behaviour are included in the argument list; a process may not use any other free names.

As well as the *process* annotation, we supply a replacement for the Haskell library function *forever*, named *foreverP* that has identical semantics, but correctly deals with the repeating behaviour in our mirror implementation. These annotations can be used with the normal library (*process* becomes benign, *foreverP* is simply defined as *forever*) without effect, but then they take on significance in the mirror implementation.

5 IO Actions

An interaction with the program's external environment is an action in the IO monad in Haskell. This can include (but is not limited to) interactions with files, sockets and graphical user interfaces. Results of these interactions cannot be predicted and are non-deterministic. Our generated model accounts for this by modelling the interactions as events selected by the environment. The complication is modelling the value returned by operations such as reading from a file.

With source code analysis it would be possible to analyse how the result affects the program's subsequent behaviour. However, with our approach, the program's behaviour can only be determined by testing it with each possible value. For values with small finite domains, such as booleans, or enumerated types, this is easily achieveable. For values such as 32-bit integers, it becomes infeasible, and for values with infinite domains such as strings, it is impossible.

Our analysis of a program is therefore accurate and feasible if the IO actions return values with small finite domains, or if the values are discarded. This is often the case for IO actions: for example, printing to the screen, waiting for a specified amount of time or writing to a file all return values of the unit type, which has a domain size of one (which also means the value is not usually inspected). We can extend our approach to cover some other cases where the value is used and the domain is large; the problem of determining a program's behaviour for a range of inputs without source code analysis is identical to the problem of program testing. Several developments have been made in this field in Haskell.

Claessen and Hughes' QuickCheck generated random values of a given type and tested that specified properties held on the output [CH00]. This was made more systematic in Runciman et al.'s SmallCheck, which generated all values up to a given depth [RNL08]. The depth of a list would be its length, whereas the depth of an integer would be its absolute value. The clever extension, Lazy SmallCheck, took advantage of Haskell's laziness to specialise on demand, allowing more efficient exploration of the state space.

We use the Lazy SmallCheck library to explore a program's behaviour given the result of an IO action. Crucially, Lazy SmallCheck will efficiently detect if the value is not used to affect the program's behaviour, as the first value it will try will be $\perp$. If the program does not raise an error with this value, it must not have evaluated it, and thus it cannot have altered its behaviour based on the value.

We use Lazy SmallCheck with an arbitrary maximum depth. If the deepest value is not reached in the lazy search, we can deduce that we have covered all possible behaviours of the program.

If this is not the case, the model generated is only an approximation of the program's behaviour, and much of the advantages of the formal method are lost. This is a necessary restriction of our approach and one that we expose to the user. We deal with values returned from reading channels in the same manner as the result of IO actions.

6 Examples

We provide two examples of our system: the first is a small example demonstrating refinement checking, the second is a larger example demonstrating deadlock freedom.

6.1 Copy Buffer

A simple example of a refinement check, taken from the FDR manual [For97], is that of a copying process, specified as follows:

$$COPY = left?x \rightarrow right!x \rightarrow COPY$$

The example implementation given in the manual, in CSP, is:

$$SEND = left?x \rightarrow mid!x \rightarrow ack \rightarrow SEND$$
$$REC = mid?x \rightarrow right!x \rightarrow ack \rightarrow REC$$
$$SYSTEM = (SEND \underset{\{X\}}{||} REC) \setminus X \text{ where } X = \{mid, ack\}$$

We can create the analogue of this implementation in CHP as follows:

```
system :: forall a. Typeable a => Chanin a -> Chanout a -> CHP ()
system input output
  = do c <- newChannelWithLabel "mid"
       d <- newBarrierWithLabel "ack"
       enroll d (\d0 -> enroll d (\d1 ->
         send input (writer c) d0 <|> rec output (reader c) d1))
       return ()
  where
    send :: Chanin a -> Chanout a -> EnrolledBarrier -> CHP ()
    send = process "send" (\input mid ack ->
      do x <- readChannel input
         writeChannel mid x
         syncBarrier ack
         send input mid ack)

    rec :: Chanout a -> Chanin a -> EnrolledBarrier -> CHP ()
    rec = process "rec" (\output mid ack ->
      do x <- readChannel mid
         writeChannel output x
         syncBarrier ack x
         rec output mid ack)
```

The backslash represents a lambda, and introduces an anonymous function (e.g. `\x -> x` is the identity function). The *enroll* function takes as its arguments a barrier, and a function that itself takes an enrolled barrier and yields a CHP process. This higher-order process style is used to scope enrolling on and resigning from the barrier.

This program can be executed normally with the CHP monad as part of a larger system, or used with our new mirror library to generate a model, which results in the following:

```
channel ack
channel in
channel mid
channel out
main_0= (((send_1) [|{| ack , mid |}|] (rec_2)))
rec_2= (mid?x_2 -> out!x_2 -> ack -> rec_2)
send_1= (in?x_1 -> mid!x_1 -> ack -> send_1)
main = main_0
```

It can be seen that, a few spurious brackets and appended unique identifiers aside, this is the same as the CSP we began with – except for the hiding. We can now prove this is a refinement of the original specification by adding the following lines to the FDR script:

```
COPY = in?x -> out!x -> COPY
assert (main\{mid,ack}) [FD= COPY
assert (main\{mid,ack}) [T= COPY
```

These refinements check successfully. We were able to take a specification, implement a CHP equivalent (in this case we had some CSP for the implementation, but this was not necessary), and make a successful refinement check against the original specification.

The main manual step required was that we hid the `mid` and `ack` events. We could attempt to infer when to hide events (a difficult option), or we could add an operator in the code to hide events. For example, we could modify a line of our system process to be:

```
enroll d (\d0 -> enroll d (\d1 ->
  send input (writer c) d0 <|> rec output (reader c) d1
    <\> [c, d]))
```

6.2 Dining Philosophers

As an example of generating a CSP model from a CHP program and checking for deadlock, we use the dining philosophers problem. The code for the deadlocking dining philosophers, that can be executed normally, or used to produce traces [BS08], is as follows:

```
fork :: EnrolledBarrier -> EnrolledBarrier -> CHP ()
fork = process "fork" (\left right ->
    foreverP ((do syncBarrier left
                  syncBarrier left )
              <-> (do syncBarrier right
                      syncBarrier right )))
```

```
philosopher :: EnrolledBarrier -> EnrolledBarrier -> CHP ()
philosopher = process "philosopher" (\left right ->
  foreverP (
   do randomDelay
      syncBarrier left <|> syncBarrier right
      randomDelay
      syncBarrier left <|> syncBarrier right))
  where
    randomDelay :: CHP ()
    randomDelay = liftCHP $ (liftIO $ getStdRandom (randomR (500000, 1000000))) >>= waitFor

college :: Int -> CHP ()
college = process "college" (\nPhil ->
  withBarrierPairListWithStem nPhil "fork_left_phil" (\forkLeftChans ->
  withBarrierPairListWithStem nPhil "fork_right_phil" (\forkRightChans ->
  runParallel_ (
    [fork (fst (forkRightChans !! n)) (fst (forkLeftChans !! ((n + 1) `mod` nPhil)))
    | n <- [0 .. nPhil - 1]]
    ++
    [philosopher (snd (forkLeftChans !! n)) (snd (forkRightChans !! n))
    | n <- [0 .. nPhil - 1]]))))
```

The *process* annotations on the fork and philosopher are not strictly necessary (the use of *foreverP* catches the infinite behaviour) but help to label the processes in the generated model. The `<->` operator is external choice, while `<|>` is parallel composition (and *runParallel_* is a list version).

This real running code can then be changed to use the CHP-model library by changing a single import statement (omitted for brevity). The generated model for three philosophers is shown in figure 1. If we append the line `assert main :[ deadlock free ]` to that script, FDR produces a trace of one of the deadlocks in the system:

```
BEGIN TRACE example=0 process=0
fork_right_phil2
fork_right_phil1
fork_right_phil0
END TRACE example=0 process=0
```

7 Post-Processing

After the model for a program has been determined, it is a relatively simple matter to print it out in the machine-readable CSP_M form that the FDR model checker expects. The only current drawback is that each different combination of arguments to an annotated process will correspond to a different process in the generated output. Returning to our example of the input consuming process, the system with *blackHole c* `<|>` *blackHole d* generates:

```
channel c
channel d
blackHole_1= (c?x_1 -> blackHole_1)
```

```
blackHole_2= (d?x_2 -> blackHole_2)
main_0= (((blackHole_1) ||| (blackHole_2)))
main = main_0
```

Even though the behaviour of the two black hole processes is identical except for the channel involved, we currently generate two instantiations of the process rather than one parameterised process. In future we would like to maintain the correctness of the specification, but also reduce its verbosity by merging such processes together.

7.1 Alphabets

In CHP, the synchronisation rules are as follows. A channel requires exactly two processes to synchronise on it. A barrier has an enrollment count, and a number of processes equal to that count must synchronise on the barrier for it to complete.

In Roscoe's CSP [Ros97], an event can have any number of parties, from one upwards. Which processes must synchronise with each other on an event is determined by the shared alphabet when the two processes are composed in parallel. Given this CSP:

$$P = a \rightarrow b \rightarrow c \rightarrow \text{SKIP}$$
$$Q = b \rightarrow \text{SKIP}$$
$$R = c \rightarrow \text{SKIP}$$
$$ALL = P \underset{\{b\}}{\|} Q \underset{\{c\}}{\|} R$$

The event a will involve just P, whereas b will involve P and Q, and c will involve P and R. Should the event a have been included in the alphabet of either parallel composition, P would cause deadlock as the other process in the composition would not be offering the event a.

In translating CHP programs into CSP models, we must infer the alphabets for parallel compositions. Since all of our events have unique identifiers we are able to follow a simple rule: the events in the alphabet of parallel composition are the intersection of the sets of events engaged in by the two processes being composed.

This rule works correctly for most programs (including the dining philosophers example). However, consider the following program:

```
p = do a <- newBarrierWithLabel "a"
       b <- newBarrierWithLabel "b"
       enroll a (\a0 -> enroll a (\a1 -> enroll b (\b0 -> enroll b (\b1 ->
           runParallel [do syncBarrier a0
                           syncBarrier b0
                       , syncBarrier a1 ])))
```

This program will deadlock when run, as only one process (of two enrolled) will synchronise on the barrier b. If the specification is generated with our original simple rule, we get this specification:

```
p = (a -> b -> SKIP) [|{| a |}|] (a -> SKIP)
```

This specification will not deadlock, as the event *b* is not in the shared alphabet with any other process. More generally, if less processes are using a barrier than are enrolled, or if only one process is using a particular channel, the model generated with our simple alphabet rule will be incorrect.

The simplest solution to this is as follows. We augment our framework to track how many processes should be using a particular event (two for channels, the enrollment count for barriers). If the actual number of processes turns out to be lower than this, we compose the relevant processes in parallel with a dummy process (`SKIP`) with the events shared, for example:

```
p = ((a -> b -> SKIP) [|{| a |}|] (a -> SKIP))
      [|{| b |}|] SKIP
```

This new model will reveal the deadlock in the original program.

8 Summary

Our approach is able to generate models for the following features of CHP programs:

- sequential and parallel composition,
- external choice,
- barrier creation and synchronisations (but not a proper semantics of dynamic enrollment and resignation),
- channel outputs,
- channel inputs and lifted IO actions (where the result either has a small finite domain, or is not used to make a choice about the program's behaviour), and
- terminating pure computations.

The main area not supported is where the domain of channel-reads and IO-actions is large *and* the result is used to influence decisions about the program's flow. We also do not support programs with pure computations that do not terminate – such non-termination would also be problematic in the real running version of CHP.

9 Conclusions

We have demonstrated a technique for generating CSP models of Haskell programs that use the CHP library. It does not require complete access to the source code, and requires minimal program annotation. The technique can be used to generate models for prototype programs (or programs developed with an agile methodology) or to generate models for programs and perform a refinement check against a specification.

The CSP_M that is generated by our approach can be passed directly to tools such as FDR in order to prove properties such as freedom from deadlock or perform refinement checks. It can

also be used with other model-checkers (such as ProB [LF08]) or tools (such as FDR explorer [FW09]) on the specification. Alternatively, a tool such as ProBE [Ros97] could be used to explore the possible traces of a program by deciding which of all the available events should happen next.

9.1 Availability

The CHP library is already available for general use – more details can be found at http://www.cs.kent.ac.uk/projects/ofa/chp/. We hope to soon release the mirror implementation described in this paper that generates CSP models.

Bibliography

[Bro08] N. C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In *Communicating Process Architectures 2008*. Pp. 67–84. Sept. 2008.

[BS08] N. C. C. Brown, M. L. Smith. Representation and Implementation of CSP and VCR Traces. In *Communicating Process Architectures 2008*. Pp. 329–345. Sept. 2008.

[CH00] K. Claessen, J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 Manual*. 1997.

[FW09] L. Freitas, J. Woodcock. FDR Explorer. *Formal Aspects of Computing* 21:133–154, 2009.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. http://www.usingcsp.com/

[LF08] M. Leuschel, M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*, 2008.

[RNL08] C. Runciman, M. Naylor, F. Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*. Pp. 37–48. ACM, 2008.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997. http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf

```
channel fork_left_phil0
channel fork_left_phil1
channel fork_left_phil2
channel fork_right_phil0
channel fork_right_phil1
channel fork_right_phil2
college_1=
  (((fork_2)
    [|{| fork_left_phil1 , fork_right_phil0 |}|]
    ((fork_3)
     [|{| fork_left_phil2 , fork_right_phil1 |}|]
     ((fork_4)
      [|{| fork_left_phil0 , fork_right_phil2 |}|]
      ((philosopher_5) ||| ((philosopher_6) ||| (philosopher_7)))))))
fork_2= (repeated_8)
fork_3= (repeated_9)
fork_4= (repeated_10)
main_0= (college_1)
philosopher_5= (repeated_11)
philosopher_6= (repeated_12)
philosopher_7= (repeated_13)
repeated_8=
  (((fork_right_phil0 -> fork_right_phil0 -> SKIP)
    []
    (fork_left_phil1 -> fork_left_phil1 -> SKIP))
   ; repeated_8)
repeated_9=
  (((fork_right_phil1 -> fork_right_phil1 -> SKIP)
    []
    (fork_left_phil2 -> fork_left_phil2 -> SKIP))
   ; repeated_9)
repeated_10=
  (((fork_right_phil2 -> fork_right_phil2 -> SKIP)
    []
    (fork_left_phil0 -> fork_left_phil0 -> SKIP))
   ; repeated_10)
repeated_11=
  (((fork_left_phil0 -> SKIP) ||| (fork_right_phil0 -> SKIP))
   ; ((fork_left_phil0 -> SKIP) ||| (fork_right_phil0 -> SKIP))
   ; repeated_11)
repeated_12=
  (((fork_left_phil1 -> SKIP) ||| (fork_right_phil1 -> SKIP))
   ; ((fork_left_phil1 -> SKIP) ||| (fork_right_phil1 -> SKIP))
   ; repeated_12)
repeated_13=
  (((fork_left_phil2 -> SKIP) ||| (fork_right_phil2 -> SKIP))
   ; ((fork_left_phil2 -> SKIP) ||| (fork_right_phil2 -> SKIP))
   ; repeated_13)
main = main_0
```

Figure 1: The generated model for the deadlocking version of the dining philosophers with three philosophers.

A decidable class of verification conditions for programs with higher order store

Nathaniel Charlton and Bernhard Reus

School of Informatics
University of Sussex

Abstract: Recent years have seen a surge in techniques and tools for automatic and semi-automatic static checking of imperative heap-manipulating programs. At the heart of such tools are algorithms for automatic logical reasoning, using heap description formalisms such as separation logic. In this paper we work towards extending these static checking techniques to languages with procedures as first class citizens. To do this, we first identify a class of entailment problems which arise naturally as verification conditions during the static checking of higher order heap-manipulating programs. We then present a decision procedure for this class and prove its correctness. Entailments in our class combine *simple symbolic heaps*, which are descriptions of the heap using a subset of separation logic, with (limited use of) *nested Hoare triples* to specify properties of higher order procedures.

Keywords: higher order store, verification, nested triples, separation logic

1 Introduction

Program verification systems can be characterised according to the range and depth of properties they prove about programs, and the degree of user assistance they require. Some systems are designed for proving "full functional correctness", that is, for proving that a program performs exactly the function it is intended to. Proving full functional correctness involves a significant annotation burden, and difficult verification conditions must be discharged interactively by users due to the limitations of automated theorem proving. On the other hand systems such as ESC/Java [FLL+02] check only a limited range of properties (e.g., memory safety), which enables them to run automatically, using decision procedures or automated theorem provers. Such systems have been termed *lightweight*, and behave much like enhanced type checkers.

Among the systems for lightweight verification are several successful tools which use *separation logic* [Rey02], an extension of classical logic which facilitates local reasoning about heap-manipulating programs. These tools include Smallfoot [BCO05a] and its descendants and HIP [NDQC07]. Separation logic provides two main ingredients: the *separating conjunction* $\star$ and the *frame rule*. The formula $P \star Q$ describes heaps which can be split into two disjoint parts, one part satisfying P and the other satisfying Q. Other new subformulae include *emp*, which describes the empty heap, and the *points-to* formulae such as $x \mapsto \{\!|f_1 : y, f_2 : \_|\!\}$ which describes a heap containing exactly one object, pointed to by variable x; this object has at least the field f_1, with value y, and the field f_2 which may have any value. The frame rule states that if a command C satisfies the triple $\{P\}C\{Q\}$, and does not modify any variables free in R, then it also satisfies

$\{P \star R\}C\{Q \star R\}$. This embodies the idea that parts of the heap which the command does not access (here described by R) remain unchanged.

We work towards extending these lightweight program checking tools to higher order languages, and in particular those with higher order store. A programming language is called *higher order* if it treats procedures as first class values, so that procedures can be returned by expressions, passed as parameters etc. and then invoked later when needed. Additionally, such a language is said to have *higher order store* if procedures can be stored in updateable variables. Higher order store presents a serious obstacle to verification: since the procedure referred to by a particular name (e.g. stored in a particular variable) can change as the program executes, it is no longer adequate to use a single static specification for each procedure. *Nested triples* (e.g. [SBRY09, HYB05]) are a natural idea for reasoning about such procedure updates. In this approach, specifications (Hoare triples) for procedures can appear as ordinary assertions about program state, and hence triples can be *nested inside other triples*, in the pre- or post-conditions.

Our contribution in this paper is to give a decision procedure for a class of verification conditions (VCs) which supports the combined use of the two logical features we have described, separation logic and nested triples. By providing these features, our class includes VCs which arise naturally during the verification of heap-manipulating imperative programs with higher order store. We prove the correctness of our decision procedure, and illustrate by example the role it plays within the static program checker we are developing. The class of VCs we decide is strongly restricted; however the result we give is, to our knowledge, the first decidability result for verification conditions involving nested triples.

1.1 Related work

We have already mentioned lightweight tools which check properties of programs specified using separation logic. Decision procedures for fragments of separation logic are known, and used in such tools; [BIP08] gives arguably the most general of these. None of the existing tools allow the use of nested triples, however. Our aim is to use our decision procedure and its future extensions to extend such tools to languages with higher order store.

Various works build verification systems on top of powerful theorem proving environments such as Isabelle and Coq, which can be used for proving full functional correctness. [MN05] uses higher order logic to reason about the heap, and formalises (imperative) programs and proofs about them in Isabelle. [ZKR08] also uses higher order logic and Isabelle. We believe that these two works could be extended to address higher order store. [NMB08] shows using Coq that separation logic reasoning principles are admissible in Hoare Type Theory, an extension of a dependently and polymorphically typed functional language (allowing higher order functions) with Hoare-style specifications for state. We expect that our results, or adaptions thereof, can also be usefully integrated with such systems: decision procedures can be used to automatically discharge many proof obligations, leaving only the most involved proofs to be done interactively. The use of decision procedures in this way is particularly explicit in [ZKR08].

Several papers have investigated the theoretical aspects of adapting separation logic for use with higher order programs. [SBRY09, BRSY08, RS06, HYB05] study programs with higher order store and use logics with nested triples, while [BTY06, BY07] study higher order procedures in the style of Idealised Algol and use a form of *specification logic*. [KAB+09] also uses a

specification logic, demonstrating correctness proofs for generic implementations of some OOP design patterns. This existing research does not discuss the issue of decidability i.e. the issue of which kinds of verification conditions can be proved or falsified fully automatically.

2 An example verification problem with higher order store

Fig. 1 gives an imperative program which uses higher order store to implement (the idealised outline of) a server for a chatroom. In addition to three fixed procedures INCR, APPLY2 and MAIN, the program uses dynamically created procedures as follows. Each connected user is encapsulated in a procedure which, when invoked on a message argument m, sends m to the appropriate user. The server maintains a procedure *SendToAll* which is run whenever a connected user types a message (a 'Talk' event). *SendToAll* broadcasts its message argument to all connected users, by invoking the procedure encapsulating each user, and also updates a counter which records the number of messages sent. When a new user joins the chatroom (a 'Connect' event) the program *updates the procedure SendToAll*, replacing it with one which additionally broadcasts to the new user. The new *SendToAll* is built from the old one, the procedure encapsulating the new user, and the higher order procedure APPLY2. Using procedures like APPLY2 in a Curried style provides a useful way to dynamically generate new code; as in functional languages we write (partial) application of (Curried) procedures simply as juxtaposition.

Using nested triples and separation logic we can reason naturally about the example program. For instance, the initialisation statement "*SendToAll* := INCR *ctr*" in Fig. 1 satisfies the triple

$$\left\{\begin{array}{c}\forall x \forall y \{x \mapsto \{\!|cnt : \_|\!\}\} \\ \text{INCR } x\ y \\ \{x \mapsto \{\!|cnt : \_|\!\}\}\end{array}\right\}\ \begin{array}{c}SendToAll := \\ \text{INCR } ctr\end{array}\ \left\{\begin{array}{ccc}\forall x \forall y \{x \mapsto \{\!|cnt : \_|\!\}\} & & \forall y \{ctr \mapsto \{\!|cnt : \_|\!\}\} \\ \text{INCR } x\ y & \wedge & SendToAll\ y \\ \{x \mapsto \{\!|cnt : \_|\!\}\} & & \{ctr \mapsto \{\!|cnt : \_|\!\}\}\end{array}\right\}$$

Here the program variable *ctr* appears free in the triple for *SendToAll*, nested in the postcondition: the value of *ctr* has been "hard-wired" into *SendToAll* via partial application of the Curried procedure INCR. As explained earlier, the specification for INCR is lightweight: specifically, the pre- and post-conditions say which objects should be in the heap at which addresses, and which fields they should contain, but not what the values of the fields should be. We remark that with higher order programs one can do almost no reasoning without using universal quantifiers, as one needs to specify how procedures behave for *all* possible invocations.

Suppose we annotate our program with specifications for the three fixed procedures, plus an invariant for the loop in MAIN, as in Fig. 2. We use the *assertion variable* $\underline{\mathrm{D}}$ to stand for whatever concrete data structure is used to store the necessary information about the connected users. Our verification of the code in Fig. 1 will not need any assumptions about this data structure, and so will be generic with respect to it; when one verifies implementations of *GetEvent* and *UserProcTemplate* (which we shall not do here), however, one will instantiate $\underline{\mathrm{D}}$ concretely.

We want our specifications checked by machine. A standard technique is to cut up the code into "straight-line" pieces, and from these and the annotations (specifications and loop invariant) generate verification conditions in the form of logical entailments. The VC generation step is not the subject of this paper: instead, we focus on the problem of how to automatically check the

```
procedure MAIN GetEvent UserProcTemplate
    new ctr;
    ctr.cnt := 0;
    SendToAll := INCR ctr;
    new eventBuffer;
    call GetEvent eventBuffer;
    t := eventBuffer.eventType;

    while t ≠ END do                          ▷ This is the main event-handling loop
        if t = TALK then                      ▷ A connected user has typed a message
            m := eventBuffer.message;
            call SendToAll m
        else if t = CONNECT then              ▷ A new user has joined the chatroom
            u := eventBuffer.newUserInfo;
            NewUserProc := UserProcTemplate u;
            OldSend := SendToAll;
            SendToAll := APPLY2 OldSend NewUserProc
        call GetEvent eventBuffer;
        t := eventBuffer.eventType

    dispose eventBuffer;
    dispose ctr

procedure INCR x y
    n := x.cnt;  x.cnt := n+1

procedure APPLY2 F G a
    call F a;  call G a
```

Figure 1: The (idealised) outline of a chatroom server, programmed using higher order store.

VCs once they have been generated. We just mention that we use forward reasoning rules akin to those of [BCO05b], rather than (backwards) weakest precondition rules.

The critical point is that for programs with higher order store the VCs contain (nested) triples. For instance (as can be shown by forward reasoning rules), after the initialisation section of MAIN the program state satisfies the formula Φ^{INIT} in Fig. 2. Thus one of the VCs is $\Phi^{\text{INIT}} \vDash \Phi^{\text{INV}}$, which corresponds to checking that the loop invariant is correctly established. Both Φ^{INIT} and Φ^{INV} contain triples for *SendToAll*. For this particular entailment, the main task is to show

$$\begin{array}{ll} & \forall x\,\{ctr \mapsto \{\!|cnt : \_|\!\}\}\ SendToAll\ x\ \{ctr \mapsto \{\!|cnt : \_|\!\}\} \\ \vDash & \forall x\,\{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}\ SendToAll\ x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\} \end{array}$$

which can be done using a frame rule to add $\underline{\mathrm{D}}$ on the left of $\vDash$. Similarly, one must prove VCs to show that the loop body preserves the invariant, e.g. one must prove that the new procedure stored in *SendToAll* during (one branch of) the loop body still meets the appropriate specification, which amounts to checking the following entailment (where Ψ is $\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}$).

$$\begin{array}{ll} & \forall x\,\{\Psi\}\ OldSend\ x\ \{\Psi\} \quad \wedge \quad \forall x\,\{\underline{\mathrm{D}}\}\ NewUserProc\ x\ \{\underline{\mathrm{D}}\} \\ \wedge & \forall a \left\{ \begin{array}{ll} & \forall x\,\{\Psi\}\ OldSend\ x\ \{\Psi\} \\ \wedge & \forall x\,\{\Psi\}\ NewUserProc\ x\ \{\Psi\} \ \wedge\ \Psi \end{array} \right\} SendToAll\ a\ \{\Psi\} \qquad (1) \\ \vDash & \forall x\,\{\Psi\}\ SendToAll\ x\ \{\Psi\} \end{array}$$

Specifications for fixed procedures

$$\forall x \forall y\ \{x \mapsto \{\!|cnt : \_|\!\}\}\ \textsc{incr}\ x\ y\ \{x \mapsto \{\!|cnt : \_|\!\}\}$$

$$\forall F \forall G \forall a \left\{ \forall x \{\Psi\}\ F\ x\ \{\Psi\} \wedge \forall x \{\Psi\}\ G\ x\ \{\Psi\} \wedge \Psi \right\} \textsc{apply2}\ F\ G\ a\ \{\Psi\}$$

$$\forall\ GetEvent\ \forall\ UserProcTemplate\ \{\Phi\}\ \textsc{main}\ GetEvent\ UserProcTemplate\ \{\underline{\mathrm{D}}\}$$

Loop invariant for loop in MAIN (Φ^{INV})

$$\begin{array}{ll} & \forall x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}\ SendToAll\ x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\} \\ \wedge & \forall a \forall x\ \{\underline{\mathrm{D}}\}\ UserProcTemplate\ a\ x\ \{\underline{\mathrm{D}}\} \\ \wedge & \forall buf\ \{\underline{\mathrm{D}} \star buf \mapsto \{\!|\,|\!\}\}\ GetEvent\ buf\ \{\underline{\mathrm{D}} \star buf \mapsto \{\!|eventType : \_\,, X|\!\}\} \\ \wedge & \underline{\mathrm{D}} \star eventBuffer \mapsto \{\!|eventType : \_\,, X|\!\} \star ctr \mapsto \{\!|cnt : \_|\!\} \end{array}$$

State after initialisation section of MAIN (Φ^{INIT})

$$\begin{array}{ll} & \forall x\ \{x \mapsto \{\!|cnt : \_|\!\}\}\ SendToAll\ x\ \{x \mapsto \{\!|cnt : \_|\!\}\} \\ \wedge & \forall a \forall x\ \{\underline{\mathrm{D}}\}\ UserProcTemplate\ a\ x\ \{\underline{\mathrm{D}}\} \\ \wedge & \forall buf\ \{\underline{\mathrm{D}} \star buf \mapsto \{\!|\,|\!\}\}\ GetEvent\ buf\ \{\underline{\mathrm{D}} \star buf \mapsto \{\!|eventType : \_\,, X|\!\}\} \\ \wedge & \underline{\mathrm{D}} \star eventBuffer \mapsto \{\!|eventType : t, X|\!\} \star ctr \mapsto \{\!|cnt : \_|\!\} \end{array}$$

where X is the two entries "*message* : _, *newUserInfo* : _", Ψ is $\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}$, and Φ is

$$\begin{array}{lll} \underline{\mathrm{D}} & \wedge & \forall buf\ \{\underline{\mathrm{D}} \star buf \mapsto \{\!|\,|\!\}\}\ GetEvent\ buf\ \{\underline{\mathrm{D}} \star buf \mapsto \{\!|eventType : \_\,, X|\!\}\} \\ & \wedge & \forall u\ \forall msg\ \{\underline{\mathrm{D}}\}\ UserProcTemplate\ u\ msg\ \{\underline{\mathrm{D}}\} \end{array}$$

Figure 2: Memory safety specifications for the fixed procedures in Fig. 1.

Intuitively the triple for *SendToAll* in the antecedent says that *SendToAll* works as required *provided* the procedures *NewUserProc* and *OldSend*, from which it was built, also behave properly. The decidable class of VCs that we identify and solve contains (1) and similar entailments.

3 Formal foundations

In this section, we formalise a simple logic for reasoning about imperative programs with higher order store, which includes nested triples and separation logic features. Our VCs will then be entailment problems between formulae of this logic. In order to give formal semantics to the logic, however, we first give an overview of the higher order programming language we work with (we lack the space to give formal definitions for the language).

3.1 Overview of our programming language with higher order store

The language we work with is very similar to that used in Fig. 1: it is a `while` language with heap manipulation statements, extended with higher order storable procedures with value param-

eters, and is interpreted within the solution to the following system of domain equations.

$$\begin{array}{lcllcl} \mathsf{Heap} & = & \mathsf{Rec}_{\mathbb{Z}_{>0}}(\mathsf{Rec}_{\mathsf{FieldN}}(\mathbb{Z})) & \mathsf{Stack} & = & (\mathsf{Var} \to \mathbb{Z}) \times (\mathsf{PVar} \to \mathsf{Proc}) \\ \mathsf{State} & = & \mathsf{Stack} \times \mathsf{Heap} & \mathsf{Cmd} & = & \mathsf{State} \rightharpoonup (\mathsf{State} + \{\mathsf{fault}\}) \\ \mathsf{Proc} & = & \mathsf{VarList} \times \mathsf{Cmd} & & & \end{array}$$

Here $\mathsf{Rec}_S(\mathsf{A})$ is the set of records with fields from set S and values from domain A. Such a record is written $\{\!|f_1 = v_1, \ldots, f_n = v_n|\!\}$ (where $f_1, \ldots, f_n \in S$ and $v_1, \ldots, v_n \in \mathsf{A}$). We overload the $\star$ symbol to mean the "union" of disjoint records: $\{\!|f_1 = v_1, \ldots, f_n = v_n|\!\} \star \{\!|f'_1 = v'_1, \ldots, f'_m = v'_m|\!\}$ is the record $\{\!|f_1 = v_1, \ldots, f_n = v_n, f'_1 = v'_1, \ldots, f'_m = v'_m|\!\}$ if $\{f_1, \ldots, f_n\} \cap \{f'_1, \ldots, f'_m\} = \emptyset$ and is undefined otherwise. We write $R[f := v]$ for the record R updated (if $f \in dom(R)$) or extended (if $f \notin dom(R)$) with value v at field f (and overload this notation for updating functions too). VarList is the set of finite lists of variables from Var or PVar, where $x : xs$ is a list with head x and tail xs, and $[]$ is the empty list.

There are two kinds of values in our language: integers and procedures. We also distinguish two kinds of program variables, integer variables in Var (written in lower case), and procedure variables in PVar (written in upper case). Program states take the form (s,t,h) where s is the integer part of the stack (or simply *integer stack*), mapping Vars to integer values, t is the procedure stack, mapping PVars to procedures, and h is the heap. Heap cells in our language are similar to objects in e.g. JavaScript, in that they are essentially associative arrays, associating integer values to field names. Thus, procedures can be stored on the stack but not on the heap.

Observe that the domain equations above are recursive to accommodate higher order store: commands map states to states but states themselves contain commands, stored in procedure variables. Commands can also produce the special value fault which indicates a low-level error such as accessing or disposing a non-allocated heap address. Commands are deterministic; in particular the language's memory allocator is deterministic. Procedures comprise a list of variables (the parameters) and a command (the body); when invoked, procedures run in a new stack with default values for all variables.

By $[\![C]\!]$ we denote the interpretation of a program statement C into Cmd. The interpretation of an expression E in stack (s,t), which gives a value in $\mathbb{Z} + \mathsf{Proc} + \{\mathsf{fault}\}$, is written $[\![E]\!]^{\mathscr{E}}_{(s,t)}$. As usual when separation logic is used, there is no expression for accessing the heap; rather, values needed from the heap must first be read into variables using the statement form $v := E.f$ which reads field f at address E. The expression $E_P\ E_A$ denotes the (partial) application of a (Curried) procedure E_P to an argument expression E_A. This provides a useful way to dynamically generate new procedures. For example, a statement "$G := F\ a$" copies the procedure stored in F into G, additionally fixing the first parameter to the *current* value of a. Future changes to the variable a will have no effect on the procedure stored in G, so programs cannot use the stack to dynamically create new recursions as in Landin's knot. The fixed procedures of a program, however, can be (mutually) recursive.

Procedures can only be invoked if they are not expecting further parameters; otherwise fault results. Inherent in our treatment of procedures is that all parameters behave as value parameters, so that from the caller's point of view procedure calls do not modify any stack variables. Integer values can be returned via the heap but procedures cannot.

$$\Phi \;::=\; E \mapsto \{|N^*|\} \mid emp \mid \underline{\mathrm{I}} \mid \Phi \star \Phi \mid \Phi \wedge \Phi \mid \forall v\, \Phi \mid \{\Phi\}\, E\, \{\Phi\} \qquad N \;::=\; f : E \mid f : \_$$

$$[\![\Phi_1 \wedge \Phi_2]\!]_\rho = [\![\Phi_1]\!]_\rho \cap [\![\Phi_2]\!]_\rho \qquad [\![emp]\!] = \{(s,t,h) \mid h = \{|\}\} \qquad [\![\underline{\mathrm{I}}]\!]_\rho = \{(s,t,h) \mid h \in \rho(\underline{\mathrm{I}})\}$$

$$[\![\Phi_1 \star \Phi_2]\!]_\rho = \{(s,t,h) \mid \exists h_1, h_2 \text{ s.t. } h = h_1 \star h_2, (s,t,h_1) \in [\![\Phi_1]\!]_\rho \text{ and } (s,t,h_2) \in [\![\Phi_2]\!]_\rho\}$$

$$[\![\forall v\, \Phi]\!]_\rho = \{(s,t,h) \mid \text{for all } n \in \mathbb{Z},\ (s[v := n], t, h) \in [\![\Phi]\!]_\rho\}$$

$$[\![E \mapsto \{|N^1, \ldots, N^k|\}]\!] = \left\{ (s,t,h) \;\middle|\; \begin{array}{l} dom(h) = \{[\![E]\!]^{\mathscr{E}}_{(s,t)}\} \text{ and for each } N_i = f : X \\ f \in dom(h([\![E]\!]^{\mathscr{E}}_{(s,t)})) \text{ and if } X \text{ is an expression} \\ \text{(i.e. not \_) then } (h([\![E]\!]^{\mathscr{E}}_{(s,t)}))(f) = [\![X]\!]^{\mathscr{E}}_{(s,t)} \end{array} \right\}$$

For $[\![\{\Phi_1\}E\{\Phi_2\}]\!]_\rho$ see Definition 1. E means any expression of the programming language.

Figure 3: Syntax and semantics for a simple logic with separating conjunction and nested triples.

3.2 A simple logic with nested triples and separating conjunction

Fig. 3 gives the syntax and semantics of our logic. Let AVar be the set of assertion variables (introduced on page 107) such as $\underline{\mathrm{D}}$; these are interpreted by an environment $\rho : \mathsf{AVar} \to \mathscr{P}(\mathsf{Heap})$. Assertions are then interpreted w.r.t. a state (s,t,h) and such a ρ. When no assertion variables are present we omit ρ. The following definition gives our total correctness, fault-avoiding interpretation of triples. This definition is non-standard in that it also requires the procedure E to behave "locally": *any* extra piece of heap h_I must be left untouched by the procedure.

Definition 1 Semantics of nested triples. We define $[\![\{\Phi_1\}E\{\Phi_2\}]\!]_\rho := S \times \mathsf{Heap}$ where $S \subseteq \mathsf{Stack}$ is all those stacks (s,t) such that:

1. $[\![E]\!]^{\mathscr{E}}_{(s,t)}$ has the form $([], c)$ where $c \in \mathsf{Cmd}$, and
2. for all disjoint heaps $h, h_I \in \mathsf{Heap}$, if $(s,t,h) \in [\![\Phi_1]\!]_\rho$ then there exist a heap g and a stack (s_L, t_L) such that $c(s^0, t^0, h \star h_I) = (s_L, t_L, g \star h_I)$ where $(s,t,g) \in [\![\Phi_2]\!]_\rho$.

Here the stack (s^0, t^0) maps all Vars to the default value 0, and maps all PVars to $([], d)$, where $d \in \mathsf{Cmd}$ always faults. The callee's local stack (s_L, t_L) is thrown away when the procedure returns and so, as stated earlier, procedure calls do not modify any of the caller's variables. □

The interpretation of the other connectives is standard. Because procedure calls do not change the value of any variables, we can quantify *program* variables over nested triples, and do not need a separate set of auxiliary variables. For a quantified formula $\Phi = \forall x_1 \ldots \forall x_n \Psi$ we will write $\Phi(E_1, \ldots, E_n)$ for the instantiation $\Psi[x_1, \ldots, x_n \backslash E_1, \ldots, E_n]$, where $E_1, \ldots, E_n$ are any expressions.

Definition 2 Entailment between assertions. Let Φ_1, Φ_2 be assertions. We say that Φ_1 *entails* Φ_2, and write $\Phi_1 \vDash \Phi_2$, if: for all $\rho : \mathsf{AVar} \to \mathscr{P}(\mathsf{Heap})$, $[\![\Phi_1]\!]_\rho \subseteq [\![\Phi_2]\!]_\rho$. □

We list the properties of entailment between triples that we will need in our proofs.

Frame property. $\{P\}E\{Q\} \vDash \{P \star R\}E\{Q \star R\}$. Because our inner triples concern procedure calls, and we use only value parameters, the usual side condition concerning variable modification [Rey02] is not needed. In our setting the frame property follows directly from the semantics of Hoare triples: the heap h_I in Definition 1 accounts for the added invariant R. This approached is called "baking in" the frame property, and is borrowed from [BY07].

AVar**-substitution property.** If $T_1 \vDash T_2$ then $T_1[\underline{\mathrm{I}} \backslash \Phi] \vDash T_2[\underline{\mathrm{I}} \backslash \Phi]$. This is a typical substitution property. (Again no side condition concerning variable modification is needed.)

Consequence property. If $P_2 \vDash P_1$ and $Q_1 \vDash Q_2$ then $\{P_1\}E\{Q_1\} \vDash \{P_2\}E\{Q_2\}$. This holds in the same way that the consequence rule for basic Hoare logic holds.

Crossing Out property. If Θ does not depend on the heap and does not contain $x_1, \ldots, x_n$ free, then $\Theta \wedge \forall x_1 \ldots x_n\{\Theta \wedge P\}E\{Q\} \vDash \forall x_1 \ldots x_n\{P\}E\{Q\}$. This follows by standard logical manipulation from the axiom (e5) in [HYB05] which also holds in our setting. This axiom states that $\Phi \rightarrow \{P\}E\{Q\}$ is equivalent to $\{\Phi \wedge P\}E\{Q\}$ provided Φ does not depend on the heap.

3.3 Simple symbolic heaps (SSHs)

We next define a simple class of formulae that we will work with, inspired by the *symbolic heaps* of [BCO05b], and set up some machinery for manipulating them.

Definition 3 Simple symbolic heaps. A *simple symbolic heap* (SSH) is a formula

$$\underline{\mathrm{I}}_1 \star \cdots \star \underline{\mathrm{I}}_k \star \bigstar_{i=1}^{n} v_i \mapsto \{\!|f_i^1 : v_i^1, \ldots, f_i^{m_i} : v_i^{m_i}|\!\}$$

where $\underline{\mathrm{I}}_1, \ldots, \underline{\mathrm{I}}_k \in \mathsf{AVar}$ are distinct, $f_i^1, \ldots, f_i^{m_i} \in \mathsf{FieldN}$ are distinct for each i, $v_1, \ldots, v_n \in \mathsf{Var}$ are distinct and each v_i^j is either an integer variable or the special symbol $\_$. (Here we allow $k = 0$ and/or $n = 0$, leaving just *emp* in the appropriate part of the formula.) Each assertion variable or points-to formula occurring in the formula is called a *spatial conjunct*. □

Definition 4 "Ensures" relation. We define a relation *ensures*, between spatial conjuncts of the kind found in SSHs, as follows.

- $u \mapsto \{\!|f^1 : u^1, \ldots, f^N : u^N|\!\}$ ensures $v \mapsto \{\!|g^1 : v^1, \ldots, g^M : v^M|\!\}$ if $u = v$ and for each entry $g^j : v^j$ there is an entry $f^l : u^l$ with $f^l = g^j$ and such that v^j is either u^l or $\_$.
- A formula $\underline{\mathrm{I}} \in \mathsf{AVar}$ ensures itself.

We say that an SSH Φ *ensures* an SSH Ψ if Φ, Ψ have the same number of spatial conjuncts, and each spatial conjunct of Ψ is ensured by a spatial conjunct of Φ.

We write $P =_{\mathrm{SSH}} Q$ when SSHs P and Q are syntactically equal (modulo the order of the spatial conjuncts, and modulo the order of fields in the points-to subformulae). We lift the $\star$ connective to give an operator on SSHs in the obvious way; note that this lifted operator is partial (e.g. $A = x \mapsto \{\!|f : y|\!\}$ is an SSH, but $A \star A$ is not). Finally we define a partial subtraction operator:

$A -_{\mathrm{SSH}} B$ is the SSH C (unique up to $=_{\mathrm{SSH}}$ when it exists) such that for some SSH B' we have $A =_{\mathrm{SSH}} B' \star C$ and B' ensures B. (Where convenient, we shall treat partial operations as though they have an option type as in ML, that is, return either Some x or None.) □

Lemma 1 Properties of SSHs.

1. *The (syntactic) relation "ensures" exactly captures the (semantic) entailment relation between SSHs, i.e.* $\Phi \vDash \Psi$ *iff* Φ *ensures* Ψ.
2. *If* $P_2 -_{SSH} P_1 = R$ *then* $P_2 \vDash P_1 \star R$. *(Implication in the other direction fails in general.)*
3. *If* P *is an SSH containing no assertion variables, and* $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ *(by which we mean that the integer stack* s *is a bijection between* Var *and* $\mathbb{Z}_{>0}$*) then there exists a heap* h *such that for all* t, $(s,t,h) \in [\![P]\!]$. □

We will use stacks $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ frequently because, as noted in 3. above, they free us from the concern that a formula such as $x \mapsto \{\!|f : \_|\!\} \star y \mapsto \{\!|f : \_|\!\}$ cannot be satisfied in any heap just because x and y contain the same address (recall that $\star$ partitions the heap into *disjoint* pieces).

3.4 SSH-triples and their properties

Next we define some classes of Hoare triples based on SSHs. These classes will be used to define our decidable class of verification conditions.

Definition 5 Suitable pairs of SSHs. A pair (P,Q) of SSHs is *suitable* if, writing P and Q as

$$P = \underline{\mathrm{I}}_1 \star \cdots \star \underline{\mathrm{I}}_k \star \bigstar_{i=1}^{n} u_i \mapsto X_i \qquad Q = \underline{\mathrm{J}}_1 \star \cdots \star \underline{\mathrm{J}}_l \star \bigstar_{j=1}^{m} v_j \mapsto Y_j$$

each v_j appears somewhere in $u_1,\ldots,u_n$, and $\underline{\mathrm{I}}_1 \star \cdots \star \underline{\mathrm{I}}_k$ is equal (modulo order) to $\underline{\mathrm{J}}_1 \star \cdots \star \underline{\mathrm{J}}_l$. (The role played by suitable SSH pairs will become clear later, when we discuss Theorem 1.) □

Definition 6 SSH-triples.

- By a *level 0 SSH-triple about procedure* F we mean a triple $\forall x_1 \ldots x_n \{P\}\ F\ vs\ \{Q\}$ where (P,Q) is a suitable pair of SSHs, $F \in \mathsf{PVar}$, $vs \in \mathsf{VarList}$ are integer variables and $x_1,\ldots,x_n \in \mathsf{Var}$ occur in vs and are distinct.
- By a *level 1 SSH-triple about* F we mean a triple $\forall x_1 \ldots x_n\ \{T_1 \wedge \cdots \wedge T_k \wedge P\}\ F\ vs\ \{Q\}$ in which (P,Q) is a suitable pair of SSHs, $F \in \mathsf{PVar}$, $vs \in \mathsf{VarList}$ are integer variables, $x_1,\ldots,x_n \in vs$ are distinct and $T_1,\ldots,T_k$ are level 0 SSH-triples about distinct procedures not including F, in which $x_1,\ldots,x_n$ do not occur free. □

At two points later we will reduce an entailment problem P_1 to a simpler entailment problem P_2. As part of proving such a reduction correct, we will take a counterexample to the entailment P_2 and construct from it a counterexample to P_1. To make this reasoning work smoothly, we will work only with counterexamples that have a particular form, as per the following definition.

Definition 7 Convenient counterexamples. Consider a non-entailment $T_1 \wedge \cdots \wedge T_k \not\models T$, where $T_1, \ldots, T_k, T$ are level 1 SSH-triples containing assertion variables $\underline{\mathrm{I}}_1, \ldots, \underline{\mathrm{I}}_n$. We say that the non-entailment has *convenient counterexamples* if, for any fresh variables $a_1, \ldots, a_n$, there exist formulae $\Phi_1, \ldots, \Phi_n$ and a state (s,t,h) such that: each Φ_i is *emp* or $a_i \mapsto \{\!|\,|\!\}$, $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$, $(s,t,h) \in [\![(T_1 \wedge \cdots \wedge T_k)[\underline{\mathrm{I}}_1, \ldots, \underline{\mathrm{I}}_n \backslash \Phi_1, \ldots, \Phi_n]]\!]$ and $(s,t,h) \notin [\![T[\underline{\mathrm{I}}_1, \ldots, \underline{\mathrm{I}}_n \backslash \Phi_1, \ldots, \Phi_n]]\!]$. □

Convenient counterexamples (which really are counterexamples due to the AVar-instantiation property) are convenient in two ways. Firstly they provide states where $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ which is useful as explained above, and secondly they provide counterexamples that contain only points-to conjuncts, with assertion variables having been eliminated by substitution. In fact, because we can eliminate assertion variables in this way, we will ignore them in our sketch proofs.

Now we come to our first theorem. We show that for any level 1 SSH-triple T about a procedure F, given *any* values for the integer variables and *any* interpretation of $\underline{\mathrm{I}}, \underline{\mathrm{J}}, \ldots$, we can construct a program for F that meets the specification T.

Theorem 1 Existence of models for level 1 SSH-triples. *Let T be a level 1 SSH-triple about F. For any s, ρ there exists t such that $(s,t,h) \in [\![T]\!]_\rho$ for all h.*

Sketch proof. Let s, ρ be arbitrary; T has the form $\forall x_1 \ldots x_n\ \{T_1 \wedge \cdots \wedge T_k \ \wedge\ P\}\ F\ vs\ \{Q\}$. Let us write P and Q as in Definition 5. We build a program $C_1; \ldots; C_n$ where each C_i deals with the subformula $u_i \mapsto X_i$ in an appropriate way, to build a heap satisfying Q. If Q contains nothing of the form $u_i \mapsto X'$ then C_i is the statement **dispose** u_i which deallocates the cell at address u_i. Otherwise, some $u_i \mapsto X'$ is present in Q and C_i writes the appropriate values into the fields of the cell at u_i, e.g. if Q contains $u_i \mapsto \{\!|f_1 : y, f_2 : z|\!\}$ then C_i will be $u_i.f_1 := y; u_i.f_2 := z$. Putting $t(F) := (vs, [\![C_1; \ldots; C_n]\!])$ we get $(s,t,h) \in [\![\forall x_1 \ldots x_n \{P\}\ F\ vs\ \{Q\}]\!]_\rho$ for all h, which suffices. □

We will depend crucially on this theorem later, because to show *non*-entailments $T_A \not\models T_B$ we generally begin with a model of T_A and modify it in some way, so that it remains a model of T_A but is not a model of T_B. Note that the restrictions in Definition 5 are carefully chosen to make this theorem work: for instance, the triple $\{a \mapsto \{\!|f : \_\,|\!\}\}\ F\ vs\ \{b \mapsto \{\!|f : \_\,|\!\}\}$, in which the pair of SSHs is not suitable, is *not* satisfiable with respect to every s, but only when $s(a) = s(b)$. Intuitively this is because if the program for F (which cannot change b) wishes to allocate new heap cells, it must use whatever locations the allocator chooses, and cannot *demand* to be allocated a particular address b. The suitability condition allows assertion variables $\underline{\mathrm{I}} \in \mathsf{AVar}$ to be used only as invariants, which means that the program constructed in the proof of Theorem 1 can simply leave alone the parts of the heap corresponding to assertion variables.

4 Three decidable entailment problems involving nested triples

In this, the main section of the paper, we give decision procedures for three classes of entailment problem involving triples, and prove their correctness. The classes are of increasing difficulty and each algorithm builds on the previous one.

procedure DECIDE-ENT-QF-0($\{P_A\}\ F\ vs\ \{Q_A\},\ \{P_B\}\ F\ vs\ \{Q_B\}$)
  if $P_B -_{\mathrm{SSH}} P_A = \mathsf{Some}\ R$ **then return** $(R \star Q_A$ ensures $Q_B)$
  else return *false*

procedure DECIDE-ENT-0($\forall x_1 \ldots x_n \{P_A\}\ F\ ts\ \{Q_A\},\ T_B$)
  let $\{P_B\}\ F\ vs\ \{Q_B\} = \mathrm{FRESHEN}(T_B)$ **in**
    if INSTANTIATE($\forall x_1 \ldots x_n \{P_A\}\ F\ ts\ \{Q_A\}, vs$) **has the form** Some $\{P\}\ F\ vs\ \{Q\}$ **then**
      return DECIDE-ENT-QF-0($\{P\}\ F\ vs\ \{Q\}, \{P_B\}\ F\ vs\ \{Q_B\}$)
    else return *false*

procedure DECIDE-ENT-1($T_1, \ldots, T_k,\ T$)
  let $\{P_2\}\ F\ vs\ \{Q_2\} = \mathrm{FRESHEN}(T)$ **in**
    if exists i **such that** T_i **is about** F **then**
      if INSTANTIATE(T_i, vs) **has the form** Some $\{T'_1 \wedge \cdots \wedge T'_r \wedge P_1\}\ F\ vs\ \{Q_1\}$ **then**
        if forall j DECIDE-ENT-1($T_1, \ldots, T_k, T'_j$) **then**
          return DECIDE-ENT-QF-0($\{P_1\}\ F\ vs\ \{Q_1\}, \{P_2\}\ F\ vs\ \{Q_2\}$)
        else return *false* **else return** *false* **else return** *false*

Figure 4: Procedures for deciding three increasingly complex entailment problems involving SSH-triples. For explanation of FRESHEN and INSTANTIATE see main text (pages 116 and 117).

4.1 Deciding entailments between quantifier-free level 0 SSH-triples

Fig. 4 gives a simple procedure DECIDE-ENT-QF-0 which, using the syntactic operators "ensures" and $-_{\mathrm{SSH}}$, decides entailment problems $\{P_A\}\ F\ vs\ \{Q_A\} \models \{P_B\}\ F\ vs\ \{Q_B\}$ between two quantifier-free level 0 SSH-triples. The following lemma establishes the procedure's correctness in the case that it returns *true*. The proof is easy, using the Frame and Consequence properties.

Lemma 2 *Let $T_A = \{P_A\}\ F\ vs\ \{Q_A\}$ and $T_B = \{P_B\}\ F\ vs\ \{Q_B\}$ be level 0 SSH-triples such that $P_B -_{SSH} P_A = R$ and $Q_A \star R$ ensures Q_B. Then $T_A \models T_B$.*

Proof. Using the Frame property to add R, T_A entails $\{P_A \star R\}\ F\ vs\ \{Q_A \star R\}$. This entails T_B by the Consequence axiom, provided we have $P_B \models P_A \star R$ and $Q_A \star R \models Q_B$. These two entailments follow from the hypotheses by Lemma 1, using parts 2. and 1. respectively. □

However, to justify our claim that DECIDE-ENT-QF-0 is a decision procedure, we must also show that when *false* is returned, the entailment genuinely does not hold; to do this, we construct a command for F that satisfies T_A but not T_B. It is in constructing such counterexample programs that the main work of the present paper lies. The following lemmas prove the required non-entailments and also show the existence of convenient counterexamples; these will be needed later when we use DECIDE-ENT-QF-0 as a subroutine when solving entailments with quantifiers.

Lemma 3 *Let $T_A = \{P_A\}\ F\ vs\ \{Q_A\}$ and $T_B = \{P_B\}\ F\ vs\ \{Q_B\}$ be level 0 SSH-triples such that $P_B -_{SSH} P_A$ does not exist. Then $T_A \not\models T_B$ with convenient counterexamples.*

Sketch proof. $P_B -_{\mathrm{SSH}} P_A$ does not exist, so there is some $v \mapsto \{\!|f^1 : v^1, \ldots, f^m : v^m|\!\}$ in P_A that is not ensured by any spatial conjunct of P_B. By Theorem 1 there exists (s,t,h) such that $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ and $(s,t,h) \in [\![T_A]\!]$ (and thus $[\![F\ vs]\!]^{\mathscr{E}}_{(s,t)} = ([],c)$ for some $c \in \mathsf{Cmd}$). We build a program C which tests for the presence of the missing heap cell at v. Specifically, we define C to be $\{\mathbf{var}\ x; C_1; \ldots; C_m\}$ where each statement C_j is constructed as follows.

$$C_j \quad := \quad \begin{cases} x := n.f^j & \text{if } v^j \text{ is } \_ \\ x := n.f^j;\ \mathbf{if}\ x = n^j\ \mathbf{then\ skip\ else\ dispose}\ 0 & \text{otherwise} \end{cases}$$

Here n is a literal integer constant with the value $s(v)$, which we can use because we work w.r.t. a fixed s. Similarly each n^j is the literal integer constant $s(v^j)$. Now we "prepend" our program C to c: put $\hat{t} := t[F := ([], c \circ [\![C]\!])]$. It suffices to show $(s,\hat{t},h) \in [\![T_A]\!]$ and $(s,\hat{t},h) \notin [\![T_B]\!]$. $(s,\hat{t},h) \in [\![T_A]\!]$ follows from $(s,t,h) \in [\![T_A]\!]$ because the precondition P_A makes sure that everything "tested" by C is in place; the call to F will behave as c. But $(s,\hat{t},h) \in [\![T_B]\!]$ fails because we can choose a heap h' which satisfies P_B but does not have the "tested" cell in place; C will fault on this heap. □

Lemma 4 *Let $T_A = \{P_A\}\ F\ vs\ \{Q_A\}$ and $T_B = \{P_B\}\ F\ vs\ \{Q_B\}$ be level 0 SSH-triples such that $P_B -_{SSH} P_A = R$ and $R \star Q_A$ does not ensure Q_B. Then $T_A \nvDash T_B$ with convenient counterexamples.*

Sketch proof. $R \star Q_A$ not ensuring Q_B can happen in several ways; we sketch the case where a whole heap cell from Q_B is missing from $R \star Q_A$, i.e. Q_B contains some $v \mapsto X$ but $R \star Q_A$ contains no $v \mapsto X'$. (Other cases include e.g. when only a field, rather than a whole cell, is missing.)

By Theorem 1 there exists (s,t,h) such that $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ and $(s,t,h) \in [\![T_A]\!]$ (and thus $[\![F\ vs]\!]^{\mathscr{E}}_{(s,t)} = ([],c)$ for some $c \in \mathsf{Cmd}$). It suffices to show $(s,t,h) \notin [\![T_B]\!]$. Since $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ there exists a heap h' such that $(s,t,h') \in [\![P_B]\!]$. From $P_2 -_{\mathrm{SSH}} P_1 = R$ and Lemma 1 we have $P_2 \vDash P_1 \star R$, so there exist disjoint h_1, h_2 such that $(s,t,h_1) \in [\![P_A]\!]$, $(s,t,h_2) \in [\![R]\!]$ and $h_1 \star h_2 = h'$. Now, $c(s^0,t^0,h') = c(s^0,t^0,h_1 \star h_2) = (s_L,t_L,h'' \star h_2)$ where $(s,t,h'') \in [\![Q_A]\!]$. Thus $(s,t,h'' \star h_2) \in [\![Q_A \star R]\!]$; since $Q_A \star R$ contains no $v \mapsto X'$ it follows that $s(v) \notin dom(h'' \star h_2)$. Suppose for a contradiction that $(s,t,h) \in [\![T_B]\!]$. Then, since $(s,t,h') \in [\![P_B]\!]$, we can use the triple T_B to see that $c(s^0,t^0,h') = (s_L,t_L,g)$ where $(s,t,g) \in [\![Q_B]\!]$. Thus $s(v) \in dom(g)$. But $g = h'' \star h_2$, so $s(v) \in dom(h'' \star h_2)$ and we have a contradiction. □

Corollary 1 **Correctness of the procedure** DECIDE-ENT-QF-0 *(Fig. 4). The procedure* DECIDE-ENT-QF-0 *decides $T_A \vDash T_B$ for quantifier-free level 0 SSH-triples T_A, T_B. Furthermore, when $T_A \nvDash T_B$ there are convenient counterexamples.*

4.2 Deciding entailments between level 0 SSH-triples

We now extend our methods to decide entailments between pairs of level 0 SSH-triples. Universal quantifiers appearing on the right of $\vDash$ are unproblematic: one can use the standard technique of replacing the quantified variables with fresh free variables. We will assume that this is done by a procedure FRESHEN. This leaves an entailment problem $T_A \vDash T_B$ of the form

$$\forall x_1, \ldots, x_n \{P_A\}\ F\ ts\ \{Q_A\} \quad \vDash \quad \{P_B\}\ F\ vs\ \{Q_B\} \tag{2}$$

between level 0 SSH-triples. The universal quantifiers appearing on the left of $\vDash$ are more difficult. The standard way to use a universally quantified formula $\forall x\Phi$ in a proof is to instantiate it i.e. choose an expression E and then make use of $\Phi[x\backslash E]$. In general, proving an entailment $\forall x\Phi \vDash \Psi$ might need us to instantiate x with several different expressions, and even where one instantiation suffices, finding the right E can be difficult.

However, recall that our use of quantifiers is quite restricted: as per Definition 6, each quantified variable x must appear as a parameter in the procedure call F *ts*. Thus the natural approach is to choose instantiations, if any exist, which "unify" F *ts* with F *vs*. In our restricted setting, this method turns out to suffice, as the following results show.

Lemma 5 *If there do not exist* $v_1,\ldots,v_n \in \mathsf{Var}$ *such that* $ts[x_1,\ldots,x_n\backslash v_1,\ldots,v_n] = vs$*, or if such exist but* $T_A(v_1,\ldots,v_n)$ *is not an SSH-triple, then (2) fails with convenient counterexamples.*

Theorem 2 *Let* $v_1,\ldots,v_n \in \mathsf{Var}$ *be such that* $ts[x_1,\ldots,x_n\backslash v_1,\ldots,v_n] = vs$ *and* $T_A(v_1,\ldots,v_n)$ *is an SSH-triple. If* $T_A(v_1,\ldots,v_n) \vDash T_B$ *fails with convenient counterexamples then so does* $T_A \vDash T_B$*.*

Sketch proof. We assume $T_A(v_1,\ldots,v_n) \nvDash T_B$ with convenient counterexamples and prove $T_A \nvDash T_B$ with convenient counterexamples, by constructing a program for F that satisfies T_A but not T_B.

Because $T_A(v_1,\ldots,v_n) \nvDash T_B$ with convenient counterexamples, there exists (s,t,h) such that $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ and $(s,t,h) \in [\![T_A(v_1,\ldots,v_n)]\!]$ (and thus $t(F) = ([p_1,\ldots,p_m],c)$ for some $c \in \mathsf{Cmd}$) but $(s,t,h) \notin [\![T_B]\!]$. By Theorem 1 there also exists t' such that $(s,t',h) \in [\![T_A]\!]$ where $t'(F) = ([p_1,\ldots,p_m],c')$ for some command $c' \in \mathsf{Cmd}$. Thus, we have two models: one for T_A, and another for $T_A(v_1,\ldots,v_n) \wedge \neg T_B$. Our idea is to merge these into a model for $T_A \wedge \neg T_B$. Here it is crucial that Theorem 1 works for *any* integer stack: if the two models had different integer stack components, the merging would not be possible. We define $\hat{t} := t[F := ([],d)]$ where the command d examines its parameters and decides whether to behave as c or as c'.

$$d(S,T,H) := \begin{cases} c(S,T,H) & \text{if } S(p_1),\ldots,S(p_m) = s(v_1),\ldots,s(v_m) \\ c'(S,T,H) & \text{otherwise} \end{cases}$$

It suffices to show $(s,\hat{t},h) \in [\![T_A]\!]$ but $(s,\hat{t},h) \notin [\![T_B]\!]$. Firstly we explain why $(s,\hat{t},h) \in [\![T_A]\!]$. If the parameters do not match $s(v_1),\ldots,s(v_m)$ then d behaves like c' and the result follows from $(s,t',h) \in [\![T_A]\!]$; if the parameters do match, one combines $p_1 = s(v_1),\ldots,p_m = s(v_m)$ and $(s,t,h) \in [\![T_A(v_1,\ldots,v_m)]\!]$ to obtain the required result. Secondly, $(s,\hat{t},h) \notin [\![T_B]\!]$ follows from $(s,t,h) \notin [\![T_B]\!]$ because if the parameters match $s(v_1),\ldots,s(v_m)$ then d behaves like c. □

The other direction, if $T_A(v_1,\ldots,v_n) \vDash T_B$ then $T_A \vDash T_B$, is obvious. Informally the preceding theorem shows that when an instantiation is discovered, it is *the* correct instantiation. Identifying the appropriate $v_1,\ldots,v_n$ is a straightforward unification problem, so let us assume that a procedure INSTANTIATE does this, returning either Some T where T is the triple T_A with the appropriate instantiation already performed, or None. Armed with INSTANTIATE we can now give, in Fig. 4, an algorithm DECIDE-ENT-0 which decides entailments between level 0 SSH-triples.

Corollary 2 **Correctness of the procedure** DECIDE-ENT-0 *(Fig. 4). The procedure* DECIDE-ENT-0 *decides* $T_A \vDash T_B$ *for level 0 SSH-triples* T_A, T_B*. Furthermore, when* $T_A \nvDash T_B$ *there are convenient counterexamples.*

We emphasise that in general, instantiating quantified parameters in this way is insufficient. Given triples $T_1 = \forall x\{true\}\ F\ x\ \{a \neq 1 \vee x = 2\}$ and $T_2 = \{true\}\ F\ b\ \{a \neq 1 \vee b \neq 2\}$ we have $T_1 \vDash T_2$, but the obvious instantiation $T_1(b)$ does not entail T_2. Non-intuitively one does have $T_1(a) \wedge T_1(b) \vDash T_2$, however. The existence of such cases is what makes Theorem 2 non-obvious and non-trivial.

On the other hand, the method used to prove Theorem 2 does extend to some more general cases. Consider for instance entailment problems of the form $\forall xy\{x \mapsto \{\!|f : y|\!\} \star P_A\}\ F\ x\ \{Q_A\} \vDash \{a \mapsto \{\!|f : b|\!\} \star P_B\}\ F\ a\ \{Q_B\}$. Here y (resp. b) does not appear as a parameter, but the procedure F can safely access y (resp. b) nevertheless, by looking into the heap at x (resp. a). Thus, we can construct a body for F that tests y against a constant value, and the technique used to prove Theorem 2 still works. Hence we can conclude that the only useful instantiation for y is b.

4.3 Deciding level 1 entailment problems

Finally we are in a position to address *level 1 entailment problems*, that is, entailments which have the form $T_1 \wedge \cdots \wedge T_k \vDash T$ where $T_1, \ldots, T_k$ are level 1 SSH-triples about distinct procedures and T is a level 0 SSH-triple. Suppose we wish to prove something of the following form:

$$\forall x\{R\}\ G\ x\ \{S\}\ \wedge\ \left\{\forall x\{R'\}\ G\ x\ \{S'\}\ \wedge\ P\right\}\ F\ y\ \{Q\} \quad \vDash \quad \{P'\}\ F\ y\ \{Q'\}$$

Here we need to prove that $F\ y$ has some desired behaviour (unconditionally), but the specification we have available for $F\ y$ (on the left of $\vDash$) only applies *provided* G satisfies a particular specification $\forall x\{R'\}\ G\ x\ \{S'\}$. Our idea is to split our argument into subproofs: 1. $\forall x\{R\}\ G\ x\ \{S\} \vDash \forall x\{R'\}\ G\ x\ \{S'\}$ and 2. $\{P\}\ F\ y\ \{Q\} \vDash \{P'\}\ F\ y\ \{Q'\}$. If these two subproofs can be done, then we can use the Crossing Out property to combine them into the required proof.

The algorithm DECIDE-ENT-1, given in Fig. 4, makes precise this idea. In general, part 1. of the proof attempt can require further splitting, which leads to a recursive algorithm. The following theorem shows that this approach is both sound when it succeeds, and complete, so that when the approach fails, a counterexample to the entailment exists.

Theorem 3 *Let $\forall x_1 \ldots x_n\ \{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ ts\ \{Q_A\}$ be a level 1 SSH-triple, let $T_1, \ldots, T_k$ be a list of level 1 SSH-triples for distinct procedures not including F and let $\{P_B\}\ F\ vs\ \{Q_B\}$ be a level 0 SSH-triple. Consider the following entailment problems.*

$$\text{(A)} \quad T_1 \wedge \cdots \wedge T_k \vDash T'_1 \wedge \cdots \wedge T'_r \qquad \text{(B)} \quad \forall x_1 \ldots x_n\{P_A\}\ F\ ts\ \{Q_A\} \vDash \{P_B\}\ F\ vs\ \{Q_B\}$$

$$\text{(C)} \quad T_1 \wedge \cdots \wedge T_k \wedge \forall x_1 \ldots x_n\{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ ts\ \{Q_A\} \quad \vDash \quad \{P_B\}\ F\ vs\ \{Q_B\}$$

Then: 1. If (A) fails with convenient counterexamples, then so does (C). 2. If (A) holds and (B) fails with convenient counterexamples, then (C) fails with convenient counterexamples. 3. If (A) and (B) hold then (C) holds.

Sketch proof. For 1.: There exists a witness (s,t,h), where $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$, for the non-entailment $T_1 \wedge \cdots \wedge T_k \nvDash T'_1 \wedge \cdots \wedge T'_r$. We make use of (s,t,h) to construct a countermodel in which the antecedent of (C) is true but the consequent is false: specifically, our countermodel is $(s,\hat{t},h')$ where $\hat{t} = t[F := (ts, [\![\textbf{dispose } 0]\!])]$.

To see that the consequent triple is false in $(s,\hat{t},h')$ observe that there exists a heap h' such that $(s,t,h') \in [\![P_B]\!]$ (this follows from $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$), but running F always causes a fault. Now we show that the antecedent is true in $(s,\hat{t},h')$. The triples $T_1 \wedge \cdots \wedge T_k$ are not "about" F, so their truth is not broken by the "update" we have made at F. It remains to show that $(s,\hat{t},h) \in [\![\forall x_1 \ldots x_n \{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ ts\ \{Q_A\}]\!]$; this follows from three facts: $(s,t,h) \notin [\![T'_1 \wedge \cdots \wedge T'_r]\!]$, $T'_1,\ldots,T'_r$ are not about F, and $x_1,\ldots,x_n$ do not appear free in $T'_1,\ldots,T'_r$.

For 2.: There exists a witness (s,t,h) for $\forall x_1 \ldots x_n \{P_A\}\ F\ ts\ \{Q_A\} \not\vDash \{P_B\}\ F\ vs\ \{Q_B\}$ where $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$. Again we use (s,t,h) to construct a countermodel for the entailment (C). By iterated application of the construction in the proof of Theorem 1 there exists t', agreeing with t on F, such that $(s,t',h) \in [\![T_1 \wedge \cdots \wedge T_k]\!]$. Thus we have $(s,t',h) \in [\![T_1 \wedge \cdots \wedge T_k \wedge \forall x_1 \ldots x_n \{P_A\}\ F\ vs\ \{Q_A\}]\!]$. But we also have $(s,t',h) \notin [\![\{P_B\}\ F\ vs\ \{Q_B\}]\!]$ and this suffices.

For 3.: We use the Crossing Out property to derive

$$\begin{array}{ll} & T_1 \wedge \cdots \wedge T_k \ \wedge\ \forall x_1 \ldots x_n\ \{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ ts\ \{Q_A\} \\ \vDash & T'_1 \wedge \cdots \wedge T'_r \ \wedge\ \forall x_1 \ldots x_n\ \{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ ts\ \{Q_A\} \\ \vDash & \forall x_1 \ldots x_n\ \{P_A\}\ F\ ts\ \{Q_A\} \ \vDash\ \{P_B\}\ F\ vs\ \{Q_B\} \end{array}$$ □

Corollary 3 Correctness of the procedure DECIDE-ENT-1 *(Fig. 4). The procedure* DECIDE-ENT-1 *decides entailments $T_1 \wedge \cdots \wedge T_k \vDash T$ where $T_1,\ldots,T_k$ are level 1 SSH-triples about distinct procedures and T is a level 0 SSH-triple.*

Sketch proof. The first thing we must show is that $T_1 \wedge \cdots \wedge T_k \vDash T$ iff $T_1 \wedge \cdots \wedge T_k \vDash \text{FRESHEN}(T)$; this is a standard result. Then we need to prove that the entailment fails (with convenient counterexamples) if none of the triples $T_1,\ldots,T_k$ is about F; this is straightforward.

Next, we consider the call to INSTANTIATE. If this returns None, we can show that the entailment fails: Lemma 5 supplies the main part of the argument. On the other hand if an instantiation is returned we use Theorem 2 to show that it is *the* correct instantiation.

To finish, we use Theorem 3. If one of the calls to DECIDE-ENT-1 returns false, then part 1 of Theorem 3 applies; if all those calls return true, then parts 2 and 3 of Theorem 3 reduce the problem to a level 0 entailment problem, which Corollary 2 tells us is correctly solved. □

5 Conclusions

Revisiting our example. To demonstrate the working of our decision procedure, we revisit the verification condition (1) (on page 108) which arose from the example program in our introduction. Running on (1), DECIDE-ENT-1 first invokes FRESHEN which rewrites the desired conclusion to $\{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}\ SendToAll\ z\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}$. Then the third triple on the left of $\vDash$ is chosen and INSTANTIATE determines that a should be instantiated with z. Now two recursive calls are made to DECIDE-ENT-1, to check the two entailments

$$\begin{array}{ll} & \forall x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}\ OldSend\ x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\} \\ \vDash & \forall x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}\ OldSend\ x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\} \end{array}$$

$$\begin{array}{ll} & \forall x\ \{\underline{\mathrm{D}}\}\ NewUserProc\ x\ \{\underline{\mathrm{D}}\} \\ \vDash & \forall x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}\ NewUserProc\ z\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\} \end{array}$$

We trace into the second of these which, after dealing with the quantifiers, makes a call

$$\text{DECIDE-ENT-QF-0}\left(\begin{array}{l}\{\underline{\mathrm{D}}\}\ \mathit{NewUserProc}\ z\ \{\underline{\mathrm{D}}\},\\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\}\ \mathit{NewUserProc}\ z\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\}\end{array}\right)$$

Inside this call, $R = \underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\} -_{\mathrm{SSH}} \underline{\mathrm{D}}$ is computed and found to be $\mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}$. Then the algorithm checks that $R\star\underline{\mathrm{D}}$ ensures $\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}$ which is the case. Returning to the outer call to DECIDE-ENT-1, the final check is

$$\text{DECIDE-ENT-QF-0}\left(\begin{array}{l}\{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\}\ \mathit{SendToAll}\ z\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\},\\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\}\ \mathit{SendToAll}\ z\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\}\end{array}\right)$$

Future work. A natural next step for this line of work is to investigate which of the restrictions we have made are necessary to obtain decidability. For instance, it appears that our method will generalise to arbitrary nesting depth if we can handle conjunctions $T_1 \wedge \cdots \wedge T_k$ containing multiple triples about the same procedure. Of particular interest is to see whether one can allow, without breaking decidability, a limited use of of a universal quantifier $\underline{\forall}$ over assertion variables, which allows much more generic specifications. The specification we gave for APPLY2 in Fig. 2 is generic with respect to the parameters F, G and a, but *not* with respect to the invariant that F and G must preserve, which is fixed as $\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}$. The assertion variable $\underline{\mathrm{D}}$ represents an arbitrary invariant, but it is the *same* arbitrary invariant as mentioned in the specification of MAIN. This suffices for our example, but a better specification would be:

$$\underline{\forall}\mathrm{I}\forall F\forall G\forall a\left\{\underline{\mathrm{I}}\wedge\forall x\{\underline{\mathrm{I}}\}\ F\ x\ \{\underline{\mathrm{I}}\}\wedge\forall x\{\underline{\mathrm{I}}\}\ G\ x\ \{\underline{\mathrm{I}}\}\right\}\ \text{APPLY2}\ F\ G\ a\ \{\underline{\mathrm{I}}\}$$

One sees here that $\underline{\forall}$ supports reasoning akin to that provided by the *hypothetical frame rule* described in [OYR04], but also goes beyond it: an even more general specification is

$$\underline{\forall}\mathrm{I}\underline{\forall}\mathrm{J}\underline{\forall}\mathrm{K}\forall F\forall G\forall a\left\{\underline{\mathrm{I}}\wedge\forall x\{\underline{\mathrm{I}}\}\ F\ x\ \{\underline{\mathrm{J}}\}\wedge\forall x\{\underline{\mathrm{J}}\}\ G\ x\ \{\underline{\mathrm{K}}\}\right\}\ \text{APPLY2}\ F\ G\ a\ \{\underline{\mathrm{K}}\}$$

Universally quantified assertion variables can be instantiated with arbitrary formulae — we have $\underline{\forall}\mathrm{I}\Phi \vDash \Phi[\underline{\mathrm{I}}\backslash\Psi]$ — but the issue is how one computes the "right" Ψ. Finally, we plan to extend our work to treat programs where procedures are stored on the heap as well as on the stack.

Acknowledgements: This work was supported by EPSRC grant EP/G003173/1. We thank the anonymous referees for their suggestions for improving the presentation.

Bibliography

[BCO05a] J. Berdine, C. Calcagno, P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*. Pp. 115–137. 2005.

[BCO05b] J. Berdine, C. Calcagno, P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*. Pp. 52–68. 2005.

[BIP08] M. Bozga, R. Iosif, S. Perarnau. Quantitative Separation Logic and Programs with Lists. In *IJCAR*. Pp. 34–49. 2008.

[BRSY08] L. Birkedal, B. Reus, J. Schwinghammer, H. Yang. A Simple Model of Separation Logic for Higher-Order Store. In *ICALP (2)*. Pp. 348–360. 2008.

[BTY06] L. Birkedal, N. Torp-Smith, H. Yang. Semantics of Separation-Logic Typing and Higher-order Frame Rules for Algol-like Languages. *LMCS* 2(5), 2006.

[BY07] L. Birkedal, H. Yang. Relational Parametricity and Separation Logic. In *FoSSaCS*. Pp. 93–107. 2007.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended Static Checking for Java. In *PLDI*. Pp. 234–245. 2002.

[HYB05] K. Honda, N. Yoshida, M. Berger. An Observationally Complete Program Logic for Imperative Higher-Order Functions. In *LICS*. Pp. 270–279. 2005.

[KAB+09] N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, A. Buisse. Design patterns in separation logic. In *TLDI*. Pp. 105–116. 2009.

[MN05] F. Mehta, T. Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.* 199(1-2):200–227, 2005.

[NDQC07] H. H. Nguyen, C. David, S. Qin, W.-N. Chin. Automated Verification of Shape and Size Properties Via Separation Logic. In *VMCAI*. Pp. 251–266. 2007.

[NMB08] A. Nanevski, J. G. Morrisett, L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18(5-6):865–911, 2008.

[OYR04] P. W. O'Hearn, H. Yang, J. C. Reynolds. Separation and information hiding. In *POPL*. Pp. 268–280. 2004.

[Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. Pp. 55–74. 2002.

[RS06] B. Reus, J. Schwinghammer. Separation Logic for Higher-Order Store. In *CSL*. Pp. 575–590. 2006.

[SBRY09] J. Schwinghammer, L. Birkedal, B. Reus, H. Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. In *CSL (Computer Science Logic)*. 2009. To appear.

[ZKR08] K. Zee, V. Kuncak, M. C. Rinard. Full functional verification of linked data structures. In *PLDI*. Pp. 349–361. 2008.

High-level Proofs about Low-level Programs

Holger Gast and Julia Trieflinger

Wilhelm-Schickard-Institut für Informatik
Eberhard Karls Universität Tübingen, Tübingen, Germany

Abstract: Functional verification of low-level code requires abstractions over the memory model to be effective, since the number of side-conditions induced by byte-addressed memory is prohibitive even with modern automated reasoners. We propose a flexible solution to this challenge: assertions contain explicit memory layouts which carry the necessary side-conditions as invariants. The memory-related proof obligations arising during verification can then be solved using specialized automatic proof procedures. The remaining verification conditions about the content of data structures directly reflect a developer's understanding. The development is formalized in Isabelle/HOL.

Keywords: verification of C code, pointer programs, precise memory models

1 Introduction

The full functional verification of low-level C code has recently attracted much attention (e.g. [Tuc08b, TKN07, CMST09, RH09]). The central challenge in these applications consists in proving the disjointness of memory objects in the C memory model. Unlike in strongly typed languages like Java or C#, the inequality of pointers in C does not imply the disjointness of the memory regions occupied by the referenced objects: pointer arithmetic, pointer casts, and internal pointers to struct fields allow almost arbitrary overlaps. The strategy proposed in the literature is to maintain a typed view on the untyped memory, and to reason about that typed view: Tuch et al. [Tuc08b, TKN07] employ a variant of separation logic; Cohen et al. [CMST09] maintain a set of disjoint objects in a ghost variable. For static property checking, Rakamarić et al. [RH09] use a static analysis to identify parts of C programs that obey the split-heap model [Bur72]. The actual verification can then use pointer inequality to determine disjointness.

The disjointness of regions is, however, only one aspect of reasoning about low-level programs. Another aspect concerns the invariants and side-conditions associated with data structures. For instance, allocated blocks in C are always contiguous in memory, i.e. overflow in the pointer arithmetic inside them is excluded. This property, in turn, enables pointers into an array to be compared by the less than operator. While it is always possible to augment assertions with suitable side-conditions, these need to be handled explicitly although they are, in fact, invariants that continue to hold through all operations. Reasoning about less precise memory models is more efficient partly because the invariants are implicit. We show that it is possible to associate the invariants with descriptions of low-level memory layouts instead.

The following function, which initializes raw memory, demonstrates the point.

```
void init(char *p, char *q) {
   char *r = p;
```

```
    while (r != q)
      *r++ = 0;
}
```

The natural loop invariant asserts that `r` runs between `p` and `q` and the bytes from `p` to `r` have already been initialized (`*a` denotes reading from address `a`):

$$p \le r \wedge r \le q \wedge (\forall a.\ p \le a \wedge a < r \longrightarrow (*a) = 0)$$

The crucial point to observe is that this invariant requires reasoning about pointer inequalities. To establish the invariant, $p \le q$ has to be a precondition of the `init` function. From there, the precondition needs to be propagated upward through the call hierarchy, leading to a proliferation of assertions. Furthermore, a programmer would never initialize a memory region without this property, so that the precondition appears as a merely technical necessity.

Building on the lightweight separation method [Gas08], we propose, instead, to make memory layouts explicit in assertions and to associate side-conditions with them. In the example, the layout will contain a ptr-block p q. It describes the memory region between addresses p and q and also includes the side-condition $p \le q$. Furthemore, the proposed approach simplifies proofs about layouts [Gas09] since fewer side-conditions arise (see Section 3).

The purpose of this paper is to show how side-conditions and invariants can be maintained implicitly with the memory layout, and that handling them implicitly leads to natural proof obligations. In low-level programs it then becomes straightforward to switch between typed and untyped views. Furthermore, we show that the reasoning also covers composite objects such as structs, arrays, and linked lists. The current paper thus extends the earlier work [Gas08, Gas09] to byte-addressed memory and machine-level representations of values.

The work presented is carried out in Isabelle/HOL to ensure soundness. The proofs are therefore partly interactive. However, we structure them to enable a direct comparison with automated approaches: for each algorithm, we first prove auxiliary theorems, which would become axioms in other methods. The verification itself is then essentially automatic (cf. Section 2).

Organization Section 2 gives an overview over lightweight separation. Section 3 describes the maintenance of invariants associated with memory layouts. Section 4 applies the resulting framework to two examples. Section 5 discusses related work. Section 6 concludes.

Isabelle Notation The notation of Isabelle/HOL mostly uses standard mathematical conventions. A few exceptions need to be mentioned. Functions, as usual in higher-order logic, are curried. The function type is denoted by $\Rightarrow$. Application of function f to argument a is written by juxtaposition $f\ a$. Definitions of constants are written by $\equiv$. An interval $[a,b)$ over an ordered domain is denoted by $\{a..<b\}$. For readability, we render theorems, which in Isabelle are expressed by meta-level implication $\Longrightarrow$, as inference rules.

2 An Overview of Lightweight Separation

Assertions in separation logic [ORY01] capture both the memory content and the memory layout, and thus the disjointness of memory regions, at the same time. Although fragments can be used

for automatic verification of shape-like properties [BCC+07], reasoning in separation logic in general requires extensive user interaction [App06, Tuc08b, McC09], except when algorithms are closely tied to the shape of data structures [Tue09]. Lightweight separation [Gas08] alleviates this problem by splitting the layout and the content in assertions. The layout is then used to derive the disjointness of memory regions automatically. This section gives an overview of lightweight separation. For more details, the reader is referred to [Gas08].

2.1 Language and Hoare Logic

We consider a low-level, C-like language with side-effecting expressions, an unrestricted address operator, pointer arithmetic, and pointer casts. Its operational semantics is inspired by [Nor98]. Compared with the detailed model of C given there, we make four main simplifications: expressions are evaluated left-to-right; we neglect alignment and padding; we do not support unstructured control flow (`break`, `continue`, `goto`, `switch`); finally, we do not differentiate between allocated and initialized memory [Nor98, §3.1.2].

We use a linear, byte-addressed memory model. Addresses `addr` are isomorphic to 32-bit words [Daw07]. For syntactical reasons, we define `null` as the address 0 and also use that constant in the language syntax. Memory is a partial function from allocated addresses to bytes. It is defined by the following Isabelle/HOL record:

record memory =
  m-dom :: "addr set"
  m-cnt :: "addr ⇒ byte"
  m-valid :: bool

The domain and content are given in the fields `m_dom` and `m_cnt`, respectively. The flag `m_valid` is a history variable, which records illegal accesses to unallocated memory but does not influence the semantics.

The semantics and Hoare logic frequently refer to the static context Γ. It contains the definitions of types, structs, local variables, and functions, and it is therefore required whenever the size of a type or the address of a variable is used.

The Hoare logic captures fault-avoiding partial correctness. In particular, the generated proof obligations ensure that all accesses are legal and that the memory remains valid throughout the execution. Assertions are predicates on static contexts and memory states. Side-effecting expressions are handled by Kowaltowski's approach [Kow77], in which the postconditions in Hoare-triples for expressions are assertions about the post-state and the result value. For the control structures and function calls, we follow Schirmer's formalization [Sch05].

The Hoare logic uses forward-style reasoning to emulate the proofs in separation logic: from a given pre-condition, the post-condition is computed as follows. The Hoare rules introduce into the post-condition *inverse operators*. Consider for illustration an assignment x = e, where x is a variable and e is side-effect-free. Suppose that the pre-condition is P. In a HOL formalization, P is a predicate on the context Γ and the memory state M. The generated post-condition then is as follows (where rd-var Γ x M fetches the value stored in variable x):

$$\lambda \Gamma\, M.\ \exists M'.\ P\ \Gamma\ (\text{STORE-VAR}\ \Gamma\ \text{x}\ \text{M'}\ \text{M}) \wedge \text{rd-var}\ \Gamma\ \text{x}\ M = e \qquad (1)$$

The inverse operator STORE-VAR modifies the current state M after the assignment by replacing

the region occupied by variable x with the content of the existentially quantified pre-state M'. For a general assignment e = e', where both e and e' may have side-effects, several inverse operators may be introduced. Furthermore, any access to memory generates a proof obligation that the accessed region is allocated. In summary, our rules generalize Floyd's assignment axiom $\{ P \}\, x = e \,\{ \exists x'.\, P[x'/x] \wedge x = e \}$, which handles only the variable case.

The development of the language and logic is carried out within Isabelle/HOL, such that the logic is guaranteed to be sound w.r.t. the operational semantics. Note that although the formalization uses higher-order logic, the resulting post-conditions are first-order if the preconditions are first-order: the β-redex in the post-condition effectively replaces the reference to the pre-state M by the inverse operator. The same replacement could be carried out by a first-order verification condition generator.

2.2 Removing Inverse Operators

An assertion containing inverse operators is not useful, because it refers to some existentially quantified, previous state rather than the current state. Suppose, for example, that pre-condition P in (1) contains an assertion rd-var Γ y M $= Y$, where M refers to the pre-state of the execution. The generated post-condition then contains rd-var Γ y (STORE-VAR Γ x M' M) $= Y$, where now M refers to the post-state and M' is existentially quantified. The central concern is to prove the memory region accessed by rd-var Γ y M disjoint from the region modified by STORE-VAR Γ x M' M. In that case, we can simplify the assertion to rd-var Γ y M $= Y$, which refers to the current state M, the post-state of the execution, alone.

Towards that end, lightweight separation captures memory layouts by *covers*. A cover A is a predicate on address sets, i.e. it describes memory regions. All covers appearing subsequently will be *well-formed* covers, which accept a single memory region and can therefore be used interchangeably with that region. For example, var-block Γ x is the region occupied by variable x. The disjointness operator A ‖ B accepts the union of the regions given by covers A and B, but only if these regions are disjoint. The assertion $M \blacktriangleright A$ states that A *covers* the domain, i.e. the allocated addresses, of memory M. The weaker variant $M \triangleright A$ asserts that region A is allocated in M.

With these definitions, memory layouts can be specified. For instance, a memory containing only the two disjoint variables x and y is captured by

M ▶ var-block Γ x ‖ var-block Γ y

From this assertion, the automated tactics from [Gas08] can prove the disjointness of the regions occupied by x and y, and the inverse operator in the running example can be removed.

The approach scales directly to inductively defined data-structures, arrays with dynamic size, user-defined memory layouts for special data structures, and predicates and functions accessing part of the memory. New cover constants are introduced by the command declare-cover. New memory-accessing constants are specified by declare-accessor. For each accessor, the user has to specify, in the form of a cover, the memory region that the predicate depends on. This, of course generates a proof obligation: the accessor may not read any memory outside the specified region. A tactic accesses provided by the verification environment discharges this proof obligation automatically in almost all cases.

Nested structures in memory can be *unfolded* automatically [Gas09] to prove memory regions disjoint. In this process, covers in the memory layout are replaced by more detailed descriptions of the same memory region. For instance, to prove a single array element disjoint from a field in a struct, the array is split into slices to exhibit the sought element; the struct, likewise, is replaced by the disjoint union of its fields. The required disjointness is then obvious and can be used as before.

The application of Hoare rules and the removal of inverse operators is implemented in a tactic `step`. Using the above reasoning, it thus executes the program symbolically and computes a readable post-condition for the given pre-condition. It stops after each statement to enable the user to inspect the result and to compare it with their intuition.

In summary, lightweight separation allows the layout and content of memory to be handled independently and automatically: users can state assertions about the content in classical higher-order logic. Contrary to the case of the substructural separation logic, properties about these assertions can be proven using the automated reasoners available in Isabelle/HOL. The only new obligations are to specify the memory layout and to characterize the accessed region for newly introduced functions reading from memory. The commands declare-cover and declare-accessor automate this process in most cases.

3 Invariants of Covers

The language treated in the earlier presentation [Gas08] is based on a less precise memory model that uses natural numbers for addresses and stored values. The purpose of the current paper is to show that the automation available there carries over to the precise byte-addressed memory model from Section 2.1. The main challenge is the exclusion of overflows in the address arithmetic. This section addresses the challenge by associating side-conditions with covers.

3.1 Implicit Invariants

The necessity for associating invariants with covers is best illustrated by the example of arrays. In the old memory model with infinite address space, an array slice with base p between indices i and j simply covers the addresses between its start and end, as computed by pointer arithmetic (where $\oplus_{\Gamma,t}$ denotes C-style pointer offset by a multiple of the size of type t).

$$\mathsf{array}\ \Gamma\ t\ p\ i\ j \equiv \lambda S.\ \ S = \{\ p \oplus_{\Gamma,t} i\ ..<\ p \oplus_{\Gamma,t} j\ \} \wedge\ \ i \leq j \wedge t \neq \mathsf{void}$$

The following unfolding rule, which is the basis for automated reasoning about disjointness within arrays, then splits the array slice at an intermediate index j.

$$\frac{i \leq j \quad j \leq k}{\mathsf{array}\ \Gamma\ t\ a\ i\ k = \mathsf{array}\ \Gamma\ t\ a\ i\ j \parallel \mathsf{array}\ \Gamma\ t\ a\ j\ k} \tag{2}$$

This natural and straightforward equation does not carry over directly to the more precise memory model considered here: if the pointer arithmetic overflows, the left-hand-side may become empty, while the right-hand-side consists of one empty and one non-empty memory region.

The solution is to introduce side-conditions that prevent these overflows into the definition of array above. Section 3.3 will define arrays in the new model along with other structured types. As a result, rule (2) can, again, be proven and used for unfolding.

Re-interpretations are unfoldings that do not split the covered memory region itself, but only change the cover constant describing the region. They often replace a cover with strong invariants by a cover with weaker invariants, as is the the case when converting from a typed to an untyped view. In these cases, the unfolding rules can take the form [Gas09]:

$$\frac{p_1 \cdots p_n}{\text{is-valid } A \longrightarrow A = B}$$

The predicate is-valid asserts that A accepts at least one address set, which entails that the side-conditions associated with A hold and can therefore be used in the proof of the equality.

3.2 Memory Blocks

For low-level programs, it is sometimes necessary to reason about the raw, byte-addressed memory. At this level, we work with the word representation of addresses directly. A memory region is therefore given by its start address a and its length n as a machine word:

block a n $\equiv$ (λS. S = {a .. < a $\oplus$ n} $\wedge$ a $\leq$ a $\oplus$ n)

The block cover thus accepts an address set starting from a up to a $\oplus$ n, exclusively. Herein, $\oplus$ denotes address offset in two's complement arithmetic. Overflows in the addition are excluded by the side-condition a $\leq$ a $\oplus$ n. Note that for uniformity, empty blocks with n=0 are included in the definition. Furthermore, from a $\neq$ null we can conclude that all pointers p in the block are non-null, which is important for reasoning about internal pointers to structs. Note also that it would be possible to include an additional side-condition a $\neq$ null into the definition of block to reflect the C-standard's guarantee that the address null will never be allocated.

Equivalently, a block can be delimited by its start- and end addresses, which is useful for reasoning about access by pointer arithmetic:

ptr-block p q $\equiv$ (λS. S = { p .. < q } $\wedge$ p $\leq$ q)

These definitions solve the example from Section 1: the function's precondition includes a conjunct M ▶ ptr-block p q $\| \cdots$, from which we can deduce is-valid(ptr-block p q), hence p $\leq$ q.

For high-level reasoning, we need to express assertions about typed objects. Since types in C mainly serve to determine the size of memory objects and values, we wish to capture a typed object by an untyped block where the length is the type's size. However, the sizes of types are computed as natural numbers to avoid overflows (see also [Tuc08b]). In converting the type size to a machine word expected by block (via of-nat), a large type might map to a small block size. The is-small-type predicate therefore checks that the type's size can be represented as a machine word. The definition of a typed block is then:

typed-block Γ a t $\equiv$ (λS. block a (of-nat (sz-of-ty Γ t)) S $\wedge$ is-small-type Γ t)

3.3 Structured Types

The handling of structured types in low-level settings is non-trivial [Tuc08b], because the memory model itself does not exclude aliasing: a pointer to `int` may point to a field in a struct,

which in turn may be allocated as a local variable. We now show how these challenges can be met in lightweight separation, while maintaining the natural invariants associated with the types' structure.

Structs The basic handling of structs is straightforward: a memory object of struct type S is captured directly by a typed-block Γ p S. To expose the fields contained in a struct, we introduce a constant field-block that describes a single field (where field-off and field-ty look up the offset and type, respectively, of the field in the struct's definition in Γ):

$$\text{field-block } \Gamma \text{ p t f} \equiv \text{typed-block } \Gamma \text{ (p } \oplus \text{ (of-nat (field-off } \Gamma \text{ t f))) (field-ty } \Gamma \text{ t f)}$$

Our verification environment allows the user to define new struct types in C syntax. It generates an accessor function for each field. Furthermore, the environment proves an unfolding theorem of the following form, which allows to prove disjointness of the fields.

$$\text{typed-block } \Gamma \text{ p (struct S)} = \text{field-block } \Gamma \text{ p (struct S) f}_1 \parallel \text{field-block } \Gamma \text{ p (struct S) f}_2 \parallel \cdots$$

Note that a general unfolding theorem cannot be formulated: while the side-conditions associated with the block for the entire struct imply those for the fields, the sum of the field sizes may not be representable as a machine word. The theorem can, however, be proven automatically if the struct size is known, as is the case for specific struct definitions.

Arrays Arrays, like in C, are defined by pointer arithmetic. The memory region occupied by an array slice with indices [i,j) can therefore be expressed directly using ptr-blocks, as in the first line of the following definition. The entire definition must include, however, further invariants:

$$\begin{aligned} \text{array } \Gamma \text{ t p i j} \equiv\ & \lambda \text{S. ptr-block (p } \oplus_{\Gamma,\text{t}} \text{ i) (p } \oplus_{\Gamma,\text{t}} \text{ j) S } \wedge \\ & 0 \leq_s \text{ i } \wedge \text{ i } \leq_s \text{ j } \wedge \\ & \text{unat j * (sz-of-ty } \Gamma \text{ t)} \leq \text{ unat max-word } \wedge \\ & \text{t} \neq \text{ TVoid } \wedge \text{ wf-ty } \Gamma \text{ t} \end{aligned} \tag{3}$$

The central goal is to reduce reasoning about arrays to reasoning about the index ranges that the verified program manipulates. The invariants associated with ptr-blocks are, however, too weak for this purpose, since overflows in the address arithmetic would lead to the wrong memory region. As in related work (e.g. [BPS09]), we consider arrays with non-negative indices (line 2). For these, line 3 excludes overflows in the address arithmetic. Finally, the last line adds two requirements of the C standard: the element type t is not `void` and it is well-formed, i.e. its size can be computed.

With this definition, the natural unfolding rule (2), which was used in the less precise model with an unbounded address space, continues to hold for the low-level memory model. Only the premises of the rule are now signed word comparisons. Accordingly, the automated reasoning about disjointness of array slices and array elements works as expected.

When accessing arrays by pointer arithmetic, array elements must be re-interpreted as typed blocks, which is achieved by the following theorem:

$$\text{is-valid(array } \Gamma \text{ t a j (j+1))} \longrightarrow \text{array } \Gamma \text{ t a j (j+1) = typed-block } \Gamma \text{ (a } \oplus_{\Gamma,\text{t}} \text{ j) t} \tag{4}$$

The treatment of invariants in this rule highlights our approach: while the side-conditions of the array imply those of the typed block, the other direction does not hold directly, because the

```
pre:  M ► «src : struct point» ‖ «dst : struct point» ∧ point-known Γ ∧
      src = SRC ∧ dst = DST ∧ src ⟶ x = X ∧ src ⟶ y = Y
post: M ► «SRC : struct point» ‖ «DST : struct point» ∧
      SRC ⟶ x = X ∧ SRC ⟶ y = Y ∧ DST ⟶ x = X ∧ DST ⟶ y = Y
void copy_point(struct point *src, struct point *dst) {
  int i;
  i = 0;
  [inv M ► «src : struct point» ‖ «dst : struct point» ‖ src ‖ dst ‖ i ∧
    point-known Γ ∧ 0 ≤s i ∧ i ≤s sz-of-ty Γ (struct point) ∧
    src = SRC ∧ dst = DST ∧ «src ⟶ x» = X ∧ «src ⟶ y» = Y ∧
    (∀ j. 0 ≤s j ∧ j <s i ⟶ rd Γ (dst ⊕Γ,char j) char M = rd Γ (src ⊕Γ,char j) char M)
  ]
  while (i < sizeof(struct point)) {
    *((char *)dst + i) = *((char *)src + i);
    i++;
  }
}
```

Figure 1: Byte-wise copy of structs

conjuncts in line 2–4 of (3) cannot be proven. The additional validity assumption provides the necessary information. As a result, the unfolding applies only in a context where the pointer arithmetic takes place inside an array, which is just the requirement stated in the C standard.

4 Applications

Sections 2 and 3 have introduced a verification environment for low-level programs written in a C-like language. Its foremost feature is the explicit formalization of memory layouts and a flexible mechanism for automated reasoning about the disjointness of memory regions. This section applies the environment to two examples that demonstrate the interaction of typed and untyped views on the memory and the handling of low-level data structures.

For readability, we have simplified slightly the notation of assertions, since the actual statements need to respect the restrictions of the Isabelle parser. In particular, variables in layouts denote their variable blocks and «p : t» is a typed block. In both cases, the context Γ is implicit.

4.1 Struct-Copy

The first example that we consider is copying structs by copying their byte representations. The annotated function `copy_point` is shown in Figure 1. It receives pointers to structs `src` and `dst`, which are defined by `struct point { int x; int y; }`. In the loop, these base pointers are cast into `char`-pointers, the objects are then accessed as `char`-arrays. The precondition asserts that the referenced structs are allocated and captures the (typed) initial values in auxiliary variables SRC, DST, X, and Y. The postcondition asserts that the typed view on `src` remains unchanged and that the typed content `dst` is same as that of `src`.

The purpose of the example is to demonstrate how typed and untyped views onto memory can

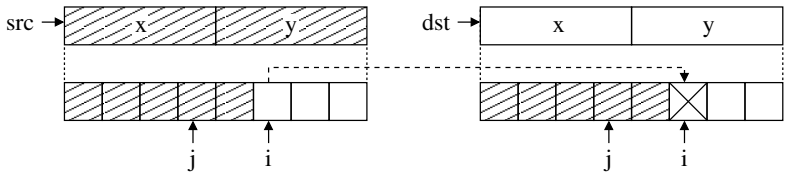


Figure 2: Memory layouts in struct copy

be handled automatically at the same time. The challenge manifests itself in the loop invariant in Figure 1. The invariant describes the memory layout, with the two structs and the local variables `src`, `dst`, and `i`. Furthermore, it limits `i` to index a byte inside the struct and asserts that the fields in `src`-struct, as well as the parameters, are unchanged. The final conjunct concerns the low-level access: the bytes between indices 0 and `i` have been already copied from `src` to `dst`.

Figure 2 shows the different levels of reasoning involved in proving the maintenance of the loop invariant. At the top, the typed views on the structs `src` and `dst` each consist of two 4-byte fields. At the bottom, the same memory regions are re-interpreted as byte arrays of size 8. The shown index i is the current position of the copying process. The depicted j ranges over the copied indices up to i. The shading indicates memory regions about which the invariant makes an assertion: the partial result of the copying process and the typed view of `src`.

Consider now the assignment in the loop body, which in Figure 2 is depicted by the dashed arrow. To show that the loop invariant is maintained, we have to prove that writing to element i, indicated by the cross, does not affect the assertions about the already copied region in the untyped view, as well as the assertion about `src->x` and `src->y` in the typed view. This is shown automatically by proving the respective memory regions disjoint as follows.

The first task is to locate the written array element i in the memory layout. Towards that end, the automatic tactics re-interpret the typed view given in the loop invariant as an array of bytes using theorem (5).

$$\frac{\text{point-known } \Gamma}{\text{typed-block } \Gamma\ p\ (\text{struct point}) = \text{array } \Gamma\ \text{char}\ p\ 0\ 8} \tag{5}$$

The tactics then identify element i by applying (2) and (4). The element j is located in the same manner. Since both elements reside in the same array, the tactics then split that array, using (2) and the upper bound on j, to prove them disjoint.

After element i has been located, the remaining proof obligations are straightforward: the fields src→x and src→y are not contained in dst, so they are obviously disjoint from element i. The same holds for element j in the untyped view of src. Since local variables are memory-allocated, incrementing i in the next statement again generates similar proof conditions, which are solved in the same way as those above.

It remains to derive the postcondition from the invariant and the negated loop test. The following theorem provides the key argument: it replaces the typed view on a memory region by its byte-representation obtained by function rd-bytes. It is then sufficient to unfold the definitions of the constants that access the fields x and y to show that the representations are equal.

$$\frac{M \rhd \text{typed-block } \Gamma\ a\ t}{\text{rd } \Gamma\ a\ t\ M = \text{rd-bytes } \Gamma\ a\ (\text{sz-of-ty } \Gamma\ t)\ M}$$

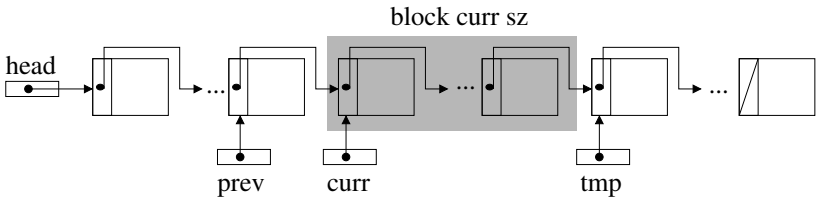


Figure 3: Allocator list structure

The example has been chosen to highlight the interaction between typed and untyped views. Of course, we can also define a function `void memcpy(void *src, void *dst, int n)` and call it with the base pointers of two `point`-structs. The frame-condition of this function asserts that `*src` will not be modified, and reasoning about the byte representation yields the same result as the specialized function above.

4.2 Low-level List Structure

As our second example, we use the `kalloc` routine of the L4 microkernel as verified by Tuch [Tuc08a, Chapter 6]. Here, we focus on the innermost loop which contains the essence of the algorithm and whose invariant subsumes that of the outer loop.

Figure 3 illustrates the idea of the algorithm. The data structure of the allocator consists of a singly-linked list of 1 kb *chunks*. The first 4 bytes of each chunk contain the pointer to the next chunk. The elements of the list are ordered by their start addresses. The variable `head` stores the pointer to the first list element. During allocation, the algorithm's outer loop [Tuc08a, Chapter 6] scans the list with a pointer `curr`. A pointer `prev` points to the preceding chunk, or initially to the variable `head`. At each element, the inner loop then tries to concatenate chunks starting at `curr` that are contiguous in memory, until the desired size has been obtained. The pointer `tmp` in this process points to the end of the already found block of size `sz` $\leq$ `size`. If insufficient contiguous space is available at `curr`, the inner loop stops and the outer loop continues its search.

The algorithm's code in Figure 4 is taken from [Tuc08a]. The variable `kfree_list` herein points to the variable `head` in Figure 3. The code deviates from the original in three minor ways: first, we rewrite the `for`-loop and `break` control structures because our language does not support them as yet. Second, to avoid reasoning about divisibility of words, we replace Tuch's counter `i` of the number of found chunks by their accumulated size `sz`. Third, our language does not support unsigned integers yet, so we work with signed `int` values throughout.

For the verification, we capture the data structure from Figure 3 by a few definitions. The cover (Section 2.2) ds-cover Γ p q M ("ds" for "data structure") contains the set of addresses occupied by all chunks in the linked list starting at p and ending at q. It enumerates that address set inductively as follows (where rd Γ p t M reads a value of type t from address p):

$$\frac{p = q \qquad S = \{\}}{\text{ds-cover } \Gamma\ p\ q\ M\ S} \qquad \frac{p \neq q \quad p \neq \text{null} \quad (\text{block } p\ 1024 \parallel \text{ds-cover } \Gamma\ (\text{rd } \Gamma\ p\ (\text{void*})\ M)\ q\ M)\ S}{\text{ds-cover } \Gamma\ p\ q\ M\ S}$$

If p = q, then the set is empty. Otherwise, the set contains a chunk and the remainder of the data

```
int sz = 1024;
tmp = *(void **)curr;
[inv M ▶ kfree-list ‖ size ‖ sz ‖ prev ‖ curr ‖ tmp
       ‖ «kfree-list:void*» ‖ ds-cover Γ (*kfree-list) null M ∧
   ds-inv Γ kfree-list M ∧ 0 ≤s size ∧
   prev ≠ null ∧ curr ≠ null ∧ *prev = curr ∧
   in-list Γ curr (*kfree-list) null M ∧
   (prev = kfree-list ∨ in-list Γ prev (*kfree-list) null M) ∧
   M ▷ «prev:void*» ‖ tmp ‖ sz ∧
   (tmp ≠ null ⟶ in-list Γ tmp curr null M ∧
                  ds-cover Γ curr tmp M = block curr sz)
]
while (tmp != null && sz < size) {
   if (tmp != (curr + sz)) {
      tmp = null;
   } else {
      tmp = *(void**)tmp;
      sz = sz + 1024;
   }
}
```

Figure 4: Inner loop of `kalloc`

structure is obtained by reading the next pointer from the beginning of p. The side-condition p≠null ensures that null is never allocatable, as requested by the C standard. Since the algorithm works with pointers into the chunk list, we define in the same way a predicate in-list Γ a p q M which checks that chunk a is reachable from p without going beyond q. Finally, the predicate ds-inv Γ kfree-list M assures that the chunks in the list are sorted by their start addresses.

With these predicates, the loop invariant in Figure 4 captures the situation from Figure 3 directly. The first conjunct describes the memory layout. It contains the local variables, the `head` variable (Figure 3) pointed to by kfree-list, and the chunks of the data-structure itself. The next three lines concern the pointers established by the outer loop: both prev and curr are not null, the pointer referenced by prev points to curr, and curr points into the chunk list. The pointer prev, on the other hand, either points to the `head` variable or into the chunk list.

The next line demonstrates the expressiveness of lightweight separation: since we do not know where exactly prev points to, it is not straightforward to show that * prev = curr is maintained despite the assignments in the loop body. Usually, a case-distinction would be required. Here, we can simply introduce a new view on the memory's layout which contains just the necessary disjointness assertions.

The last conjunct captures concisely the already found contiguous block between curr and tmp, which is shaded in Figure 3: if the search has not been interrupted by setting tmp=null, then the sequence of chunks is equivalent to a single contiguous memory block of size sz.

With this setup, the invariant is established initially from the invariant of the outer loop, which contains all but the last conjunct. The maintenance of the loop invariant is also straightforward: by only using tactic `step`, we reach the end of the loop body. Since `step` has proven that all but the last conjunct of the invariant are not affected by the loop body, they are maintained trivially. The last conjunct depends on the if-branch taken in the body. If tmp has been set to null to stop

the search, it is satisfied immediately. In the other case, we first have to show that tmp is still in the list, which is obvious by the definition of in-list. Finally, we need to join a new chunk into the already found block, which is the crucial proof obligation of the algorithm. It shows how lightweight separation can be used for manual proofs about memory layouts. We have to prove:

$$\text{ds-cover } \Gamma \text{ curr } (\text{rd } \Gamma \text{ old-tmp } (\text{void*})\ M)\ M = \text{ block curr } (\text{old-sz} + 1024) \tag{6}$$

from

$$\text{ds-cover } \Gamma \text{ curr old-tmp } M = \text{ block curr old-sz} \tag{7}$$

The idea of the proof is to split off the last block at old-tmp from cover in (6), which is possible by generic unfolding lemmata about the acyclic list structure. This yields a single element list from old-tmp to tmp, which can be converted into a block old-tmp 1024. It remains to show:

$$\text{block curr old-sz} \parallel \text{block } (\text{curr} \oplus \text{old-sz})\ 1024 = \text{block curr } (\text{old-sz} + 1024)$$

Joining the adjacent blocks on the left-hand-side finishes the proof.

In an unbound address space, the above steps would be obvious. Since addresses are words, we have to show that no overflows occur in the address arithmetic involved in the proof steps. The arising side-conditions are solved using that the chunks in the data structure are sorted by their start addresses. Nevertheless, the entire proof closely follows the intuition of the algorithm.

To assess the difference between our approach and separation logic, it is instructive to compare the invariant given in [Tuc08a, §6.5] with the one in Figure 4.[1] We keep the list structure as a whole and only express where the pointers prev, curr, and tmp are located inside the list. Tuch's invariant, in the spirit of separation logic, has to split the list structure at the elements pointed to by curr and tmp and make explicit that the three resulting list fragments are disjoint. This split generates additional proof obligations during verification since tmp is moved, while in our invariant, all but the last conjunct are maintained automatically.

If the loop finishes with tmp $\neq$ null, the algorithm removes the identified chunks between curr and tmp from the list structure by an assignment *prev = tmp; (not shown in Figure 4). In Tuch's case, the required split is already present in the loop invariant. With our approach, the removal is accomplished directly by lemmata from the list library. They allow a ds-cover to be split at points designated by the in-list assertions from the invariant, which capture the relative positions of prev, curr, and tmp.

In summary, our approach has enabled us to move many proof obligations from the verification of the specific `kalloc` algorithm into a generic library of lemmata about the allocator's data structure. They are, indeed, independent of the algorithm and follow a developer's intuition about the data structure. Since they are formulated in classical higher-order logic, their proofs can take advantage of the automatic reasoners available in Isabelle/HOL. The library with definitions, theorems, and proofs has just above 370 lines of Isabelle code. The verification of the loop itself consists in 165 lines, of which 50 are the statement of the lemma, and 19 are immediate invocations of `step`.

[1] For simplicity, we have left out those parts of the invariant which are established before the loop and are obviously not modified, such as the alignment of the block at `curr` at a multiple of the requested size.

5 Related Work

Several authors have studied the application of separation logic to low-level code. Appel and Blazy [AB07] give a logic for C-minor. Myreen and Gordon [MFG07] target machine code. This work focuses on the formalization, rather than the application, of the Hoare logic. Lin et al. [LC+07] verify a garbage collector using a separation logic. They report that a major challenge was the manipulation of separation logic assertions [LC+07, §5.2]. Their tactics are able to match equal separation conjuncts, but do not provide any unfolding. Appel [App06] and McCreight [McC09] address the support for interactive reasoning in separation logic, but do not aim at automatic reasoning.

A fragment of separation logic that focuses on the shape of data structures can be handled completely automatically, including the unfolding of recursively defined data structures and struct types [BCC+07]. The unfolding algorithm depends, however, on the fact that in any case only finitely many unfolding steps apply. Recently, it has been shown that the fragment can in some cases also accomodate reasoning about the content of data structures [Tue09, BPS09].

Tuch [Tuc08b] demonstrates how structs and fixed-size arrays can be handled in separation logic. He uses a deep embedding, i.e. he defines a datatype that captures the recursive structure explicitly. He models the semantics of C memory in detail, including alignment and padding. The automation available is limited to a tactic that unfolds the structure of types on demand. In contrast, we use a shallow embedding where new data structures can be defined in a flexible way, and rely on automated reasoning support.

Cohen et al. [CMST09] maintain the memory layout in a ghost variable. The variable at each point contains a set of disjoint objects identified by their (typed) base pointers. They introduce statements `split` and `join` that allow the programmer to expose or hide the internal structure of objects. With this setup, disjointness of memory regions can be derived from pointer inequality as in a typed memory model. They support structs, bit-fields, unions, and fixed-size arrays, but not arrays with dynamic size and no recursively defined data structures. They also rely on programmer assistance in maintaining the layout, while we automated the reasoning. Since only a single layout can be expressed, the multiple views used in Section 4 cannot be realized directly.

6 Conclusion

We have shown that it is possible to reason about low-level programs using high-level arguments and proofs that follow a developer's understanding. The basis of our approach is the automated reasoning about the disjointness of memory regions in the lightweight separation method. We have extended the existing mechanisms to low-level programs by including appropriate invariants into the definitions of memory layouts. One of the main challenges has been the finiteness of the address space, which entails the possibility of overflows in the address arithmetic. With the invariants, the theorems about the layouts become natural and intuitive.

Lightweight separation splits the assertions about the layout from assertions about the memory content. This split has proven particularly useful in low-level programs that use typed and untyped views on the memory at the same time (Section 4.1). Furthermore, it has enabled us to verify code that works on a low-level data structure without actually exposing the internal

memory layout of the data structure (Section 4.2). The arising proof obligations about the list structure have been solved by generic lemmata from a library, which is independent of a particular algorithm being verified and thus re-usable in further projects. Furthermore, the theorems reflect the intuitive understanding of lists.

The presented approach is very flexible, as it covers a wide range of data structures. C types such as structs and arrays of dynamic size are directly supported by the verification environment. Furthermore, the user can describe application-specific data structures, including recursively defined data types, in a straightforward manner. Once introduced, these memory structures are handled automatically just like the built-in ones (Section 2.2, 4.2).

There are two areas of future work: first, we plan to incorporate the details of the C memory model for bit-fields, unions, padding, and alignment. Second, we propose to pursue further the formulation of generic theorems about data structures as shown in Section 4.2: the list structure handled there is not, in principle, different from any singly-linked list. It will therefore be possible to provide a generic library of theorems about linked lists using Isabelle's locale mechanism.

Bibliography

[AB07] A. W. Appel, S. Blazy. Separation Logic for Small-Step C minor. In Schneider and Brandt (eds.), *TPHOLs*. LNCS 4732, pp. 5–21. Springer, 2007.

[App06] A. W. Appel. Tactics for separation logic. http://www.cs.princeton.edu/~appel/papers/septacs.pdf, Jan. 2006.

[BCC+07] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, H. Yang. Shape analysis of composite data structures. In *CAV 2007*. LNCS 4590. Springer, Heidelberg, 2007.

[BPS09] M. Botincan, M. Parkinson, W. Schulte. Separation Logic Verification of C Programs with an SMT Solver. In *4th International Workshop on Systems Software Verification (SSV 2009)*. ENTCS. Elsevier Science B.V., 2009.

[Bur72] R. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Stuctures. In Meltzer and Michie (eds.), *Machine Intelligence*. Volume 7. Edinburgh University Press, 1972.

[CMST09] E. Cohen, M. Moskal, W. Schulte, S. Tobies. A Precise Yet Efficient Memory Model for C. In *4th International Workshop on Systems Software Verification (SSV 2009)*. ENTCS. Elsevier Science B.V., 2009.

[Daw07] J. E. Dawson. Isabelle Theories for Machine Words. In *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)*. September 2007.

[Gas09] H. Gast. Reasoning about Memory Layouts. In Cavalcanti and Dams (eds.), *Formal Methods (FM 2009)*. LNCS. Springer, 2009. to appear.

[Gas08] H. Gast. Lightweight Separation. In Ait Mohamed et al. (eds.), *Theorem Proving in Higher Order Logics 21st International Conference, TPHOLs 2008*. LNCS 5170. Springer, 2008.

[Kow77] T. Kowaltowski. Axiomatic aproach to side effects and general jumps. *Acta Informatica* 7:357–360, 1977.

[LC+07] C.-X. Lin, Y.-Y. Chen, , L. Li, B. Hua. Garbage Collector Verification for Proof-Carrying Code. *JCST* 22(3):426–437, May 2007.

[McC09] A. McCreight. Practical Tactics for Separation Logic. In *Theorem Proving in Higher-order Logics (TPHOLs)*. 2009. (to appear).

[MFG07] M. O. Myreen, A. C. J. Fox, M. J. C. Gordon. Hoare Logic for ARM Machine Code. In Arbab and Sirjani (eds.), *FSEN*. LNCS 4767, pp. 272–286. Springer, 2007.

[Nor98] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998. Technical Report UCAM-CL-TR-453.

[ORY01] P. W. O'Hearn, J. C. Reynolds, H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*. LNCS 2142, pp. 1–19. Springer, 2001.

[RH09] Z. Rakamarić, A. J. Hu. A Scalable Memory Model for Low-Level Code. In *10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009)*. 2009. To appear.

[Sch05] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2005.

[TKN07] H. Tuch, G. Klein, M. Norrish. Types, Bytes, and Separation Logic. In Hofmann and Felleisen (eds.), *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. Pp. 97–108. Nice, France, Jan. 2007.

[Tuc08a] H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Aug 2008.

[Tuc08b] H. Tuch. Structured Types and Separation Logic. In *3rd International Workshop on Systems Software Verification (SSV 08)*. Feb. 2008.

[Tue09] T. Tuerk. A Formalisation of Smallfoot in HOL. In *Theorem Proving in Higher-Order Logics (TPHOLs 2009)*. 2009. to appear.

A workbench for preprocessor design and evaluation: toward benchmarks for parity games

Michael Huth, Nir Piterman and Huaxin Wang

Department of Computing, Imperial College London

Abstract: We describe a prototype workbench for the study of parity games and their solvers. This workbench is aimed at facilitating two activities: to aid in the design, validation, and evaluation of preprocessors for parity game solvers; and to aid in the generation of benchmark parity games that are meaningful for a wide range of solvers. Our workbench allows for easy composition of preprocessors, can populate databases with games and their meta-data, offers a query language for generating games of interest, and has already found potentially hard games.

Keywords: Parity games. Preprocessors. Benchmarks. Solvers.

1 Introduction

Parity games are determined 2-player games with memoryless winning strategies. These games are of fundamental interest in formal verification. Their natural decision problem, whether a particular node is won by a particular player, is equivalent to that of local model checking for the modal mu-calculus (whether a state s in a Kripke structure satisfies formula ϕ) [Sti95]. Therefore, any algorithm for solving parity games (referred to as "solver" subsequently) can serve as a model checker for the modal mu-calculus. Also, these decision problems therefore have the exact same complexity – whose determination is a longstanding open problem with the best known upper bound being UP $\cap$ coUP [Jur98]. Parity games (often referred to simply as "games" subsequently) have applications beyond model checking, e.g., in the synthesis of reactive systems from specifications [PR89] and in the determinization of automata [MS95]. Thus the design and evaluation of solvers is an important activity with impact beyond the area of model checking.

Formally, a parity game G is a pair $((V_G, E_G), \chi_G)$ where (V_G, E_G) is a directed graph[1] (the game graph of G) such that V_G is partioned by finite sets V_0^G and V_1^G of nodes owned by player 0 and 1, respectively; and $\chi_G \colon V_G \to \{0, 1, \dots\}$ assigns colors $\chi_G(v) < \infty$ to nodes.

We now explain how these games are played. A play in game G starts at some node v in V_G. The player who owns v then chooses some v' with (v, v') in E_G as next node. The play continues from v' in the same manner and thus generates an infinite sequence of nodes (as our game graphs have no deadlocks). Now consider the largest color k of those nodes that occur in that sequence infinitely often. If k is even, player 0 wins that play, otherwise player 1 wins it. A strategy for player σ is a partial function π from V_σ^G into V_G such that $(v, \pi(v)) \in E_G$ whenever $\pi(v)$ is defined. A play is consistent with strategy π if all choices made by player σ in that play are made according to π. Strategy π is winning at node v (for player σ) if all plays beginning in

[1] Without loss of generality, we assume that there is no v in V_G with $(v, v) \in E_G$ (no self-loops), and that for all w in V_G there is some w' with (w, w') in E_G (no deadlocks).

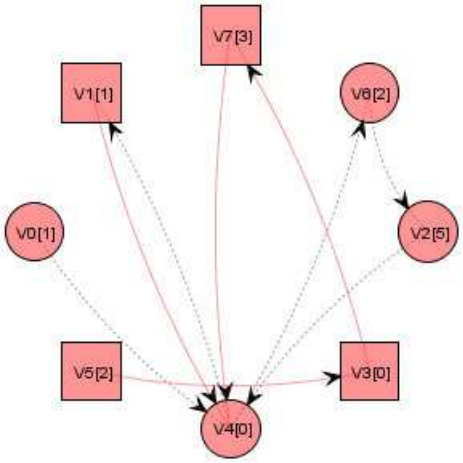


Figure 1: An 8-node parity game with 9 edges, and its solution. Nodes have canonical names, e.g. `v0`. Squared nodes are owned by player 1, circled ones are owned by player 0. Colors $\chi_G(v)$ are written within square brackets in nodes. All nodes are won by player 1, and so the solution consists only of her strategy, indicated by boldface edges.

v and consistent with π are won by player σ. A central, well-known result is that each parity game G has a partition W_0 and W_1 of V_G and both players σ have strategies π_σ winning for all nodes in W_σ (see e.g. [Zie98]). This partition and such strategies constitute what one refers to as a solution of that game.

Figure 1 shows a very simple parity game. Throughout the paper square nodes (set V_1^G) are owned by player 1, circled nodes (set V_0^G) are owned by player 0. The color $\chi_G(v)$ is written within node v in square brackets. Nodes have canonical names `v0`, `v1`, etc. Edges display E_G. Nodes colored green are won by player 0 (there are none in this game), nodes colored red are won by player 1. Boldface edges incidate moves of winning strategies. The winning strategy for player 1 moves to node `v4` whenever it can. Otherwise, it moves to `v7` or to `v3`, which it owns and from which it can move to `v7`. This strategy is winning for all nodes since it traps player 0 into cycles through `v4`, all of which player 0 loses.

The parity game in Figure 6(B) shows a non-trivial partition into winning regions. Player 0 wins nodes `v2`, `v4`, and `v7` whereas player 1 wins all other nodes.

Solvers partition, on input game G, set V_G into nodes won by player 0 and 1, respectively; and supply winning strategies for those winning regions. Existing solvers may be sub-exponential in the number of nodes (e.g. [JPZ08]) but they either have exponential worst-case running times in the index of the game G (the largest color in G plus 1: $\mathrm{index}(G) = 1 + \max_{v \in V_G} \chi_G(v)$) or it is not known whether they have polynomial running time. Given two solvers, it is not at all clear how to compare them. The worst-case input for one solver, e.g., may well be trivial as input for the other solver. And comparing solvers on a set of games is only meaningful in as much as these games can be claimed to be hard to solve for any solver. This situation is reminiscent to that of SAT solvers, where one has a set of benchmarks (formulae of propositional logic) whose satisfiability checks are known to be challenging for existing SAT solvers – e.g. a propositional

logic encoding of an elementary pigeon hole principle, that n pigeons cannot be placed into $n-1$ pigeon holes without sharing.

Another motivation for this paper is subject to future work: while current research focuses on complete algorithms (that decide the winners of all nodes), we want to consider incomplete algorithms (that decide winners of only some nodes) that work well in practice.

An, at first sight unrelated, issue is the design and evaluation of preprocessors for parity games. By a preprocessor we mean any tool that simplifies a parity game before it passes the simplified game on to a parity-game solver. The nature and extend of these simplifications can vary from trivial conversions to the solution of an "easy" part of the parity game.

One form of preprocessing is that one can transform the game G into one that is free of self-loops (edges $(v,v) \in E_G$ from a node v to itself) and deadlocks (nodes v that don't have outgoing edges), without changing the solution of the original game. This form of preprocessing is so basic, and unhelpful for the task of finding benchmarks, that we only consider games without self-loops and deadlocks in this paper. At the other end of the spectrum we have solvers, which are unhelpful for generating benchmarks that are meaningful for a whole class of solvers: a set of games that is hard for one solver may not be hard at all for another solver.

Aim of work reported here. The idea of this paper is therefore to explore the middle of that spectrum, algorithms that perform non-trivial simplifications of games and can be interpreted both as preprocessors (since they don't solve all games) and as solvers (since they may solve a substantial portion of the game, leaving a computationally hard core behind).

The overall aim of this work is therefore to develop a workbench in which an entire spectrum of such preprocessors (including solvers) can be expressed, implemented, and evaluated. This workbench is meant to support the generation of a database of games and their meta-information, so that users or automated search processes can submit queries that may return games of interest, and may validate preprocessors and solvers as well as their optimizations.

Related work. It is widely recognized that no meaningful set of benchmarks for parity games is presently available. Experimental work for solvers by and large focuses on the optimization of the underlying algorithms and their data structures. Such optimizations improve performance but make fair comparisons between solvers harder, even if good benchmarks were to be available.

The work in [ACH09] developed preprocessors `A1`, `A2`, `A3`, and `PROBE[n](A)` mentioned in this paper. But that work provided no preprocessor algebra, no query language, and no implementation work. The aforementioned preprocessors and Zielonka's solver [Zie98] were implemented as a desktop application in [Wan07], where also first statistics were run on the preprocessing of parity games.

In [FL09] the authors propose to use a generic solver that first does some preprocessing, then uses optimized solvers on special residual games (e.g. one-player games and decomposition into strongly connected components), and then only uses an input solver on residual games that cannot be further optimized. Experimental results show that this approach can procude vast speed-ups and that Zielonka's recursive algorithm [Zie98] performs surprisingly well.

Outline of paper. In Section 2 we define an algebra for composing preprocessors for parity games, impose requirements on terms from that algebra, and give some examples of preprocessors generated in that algebra. Syntax and semantics of a query language for a database of parity games are provided in Section 3. A prototype of our workbench, implementing an instance of the

above algebra and the query language, is described in Section 4. Some implementation issues are discussed in Section 5. In Section 6 we illustrate how the workbench can be used to design, validate or evaluate preprocessors – and how it can generate games of interest. Future work and our conclusions are stated in Section 7.

2 Algebra of preprocessors

We now provide a simple specification language for preprocessors that abstracts away low-level programming details and focuses on how to compose preprocessors out of more basic ones.

Algebra and its informal meaning. The preprocessors we consider are generated as regular expressions

$$p ::= a \mid p;p \mid p^+ \mid f(p) \tag{1}$$

where a ranges over a set of atomic preprocessors (which can thus accommodate any externally supplied preprocessors), $p_1;p_2$ denotes the sequential composition of preprocessors p_1 and p_2 in that order, p^+ denotes the iteration of the preprocessor p, and $f(p)$ is the "lifting" of preprocessor p by a function f. Atomicity and sequential composition are natural concepts for constructing preprocessors. The meaning of the other clauses is best explained by means of examples, where we write $\mathsf{res}(G,p)$ to denote the game output by preprocessor p on input game G, also called the *residual game of G under p*.

Two color-simplifying atomic preprocessors. Let a_1 be an atomic preprocessor that checks on game G, only once for each node v with $\chi_G(v) \geq 2$, whether there is any cycle in the game graph through v and through some node w with $\chi_G(w) = \chi_G(v) - 1$. If there is no such cycle (in particular, if there is no cycle through v at all), a_1 updates $\chi_G(v)$ by subtracting 2 from it. In the game in Figure 1, e.g., a_1 could decrement color 5 at node `v2` to 3, since there is no node with color 4 in any cycles through `v2`. Then a_1 could decrement color 2 at node `v5` to 0, since `v2` isn't on any cycle, etc.

Preprocessor a_2 similarly explores each node v with $\chi_G(v) > 0$ once. If all cycles through v have a node w with $\chi_G(v) < \chi_G(w)$, then a_2 updates $\chi_G(v)$ to 0. In the game in Figure 1, e.g., this could reset the color of node `v6` from 2 to 0, since all cycles through `v6` also go through `v2` which has color 5.

Iteration. Preprocessor a_1 is not idempotent: running it twice may get a simpler game than running it once (e.g., the first run may change a 5 into a 3, which then allows a 6 that was "blocked" by that 5 to change to 4). For input game G, preprocessor a_1^+ keeps applying a_1 until reaching a fixed point. Preprocessor a_1^+ preserves the initial game graph and terminates on all games, as seen through the well-founded ordering $G \prec_{a_1} G'$ iff $\sum_{v \in V_G} \chi_G(v) < \sum_{v \in V'_G} \chi_{G'}(v)$.

For any preprocessor p, iteration p^+ is well defined iff there is a well-founded ordering $\prec_p$ on games such that for all games G with $G \neq \mathsf{res}(G,p)$ we have $\mathsf{res}(G,p) \prec_P G$. A preprocessor p is *idempotent* iff $p;p$ and p have the same effect on all games G. Then p^+ is well defined with discrete well-founded ordering. Generally, all well defined p^+ are idempotent preprocessors.

A preprocessor using index-3 abstraction. Atomic preprocessor a_3 operates on game G as follows. It generates a sequence of index-3 games that have the same game graph as G and

whose winning regions for one player σ are also won for that player in G. Any such winning regions are deleted from G (technically, closed up under σ-attractors), and a new such sequence of index-3 games is generated on the resulting game until the game no longer simplifies. For example, if G initially has index 5, one such index-3 abstraction turns color 4 into color 2, colors 3 and 5 into 1, and all other colors into 0. Any node v won by player 1 in this modified game, can ensure that any path from v in G has either infinitely many colors 3 or 5, and only finitely many colors 4. Any such node is thus certain to be won by player 1 in G.

The game in Figure 1, e.g., is solved completely by the composed preprocessor $a_1;a_3$.

A preprocessor transformation. The lifting clause $f(p)$ has as intuition that f is a device that lifts the effectiveness of preprocessor p. We give an example, lft, such that $\mathsf{lft}(a_3)$ acts on G as follows. It considers each node v of G with at least two outgoing edges in turn: for all pairs of such outgoing edges, it creates two subgames (which implement only one of these edges and remove all other outgoing edges of v), and runs a_3 on these subgames. If a_3 decides for some node z in V_G a different winner in each subgame, node v is won in G by the player who owns it (since a_3 correctly classifies winners of deleted nodes in input games and since nodes not won by their owner cannot display such observable differences), and no further pairs of subgames for v need to be considered. Thus $\mathsf{lft}(a_3)$ also correctly classifies winners of nodes it deletes. The residual game $\mathsf{res}(G,\mathsf{lft}(a_3))$ is obtained from G by removing all nodes v (and their edges) whose winners are decided in this manner.

By induction, this is also sound for higher-order lifts $\mathsf{lft}^k(a_3)$ with $k \geq 1$, where $f^1(p)$ is defined as $f(p)$ and $f^{n+1}(p)$ as $f(f^n(p))$. We note that $f^k(p)$ generally does not have the same effect as the k-fold sequential composition of $f(p)$ with itself.

Requirements on preprocessors. Although our algebra for preprocessors is very general, we impose four requirements on all preprocessors implementable in our workbench:

1. the game graph of $\mathsf{res}(G,p)$ is a sub-graph of the game graph of G
2. the preprocessor p decides (correctly) the winners of all nodes of G that are no longer nodes in $\mathsf{res}(G,p)$
3. for each node v on the game graph of $\mathsf{res}(G,p)$, its winner is the same in both games G and $\mathsf{res}(G,p)$ and
4. for each node of $\mathsf{res}(G,p)$, its color in $\mathsf{res}(G,p)$ is no larger than its color in G.

The first requirement limits the effect that preprocessors have on the game graph to the deletion of nodes and edges. The only preprocessors that we know to violate this requirement are those that eliminate self-loops and deadlocks (which we don't consider). If one wishes, one can actually drop this requirement without affecting the overall working of our framework.

The next two requirements require little explanation: it only makes sense to remove a node from considerations when it has been decided which player wins it; and residual games have to be consistent with the original game in terms of which player wins residual nodes.

The last requirement may also be relaxed but then the iteration of preprocessors may diverge. We therefore adopt this requirement as a static constraint that, in conjunction with the other requirements, ensures that iterations converge. Specifically, preprocessors p meeting these four requirements have well defined p^+, since they all have a well founded order $G \prec G'$, defined as $\mathsf{rank}(G) < \mathsf{rank}(G')$ for the rank function $\mathsf{rank}(G) = |V_G| + |E_G| + \sum_{v \in V_G} \chi_G(v)$.

A composition pattern. We illustrate the utility of our algebra for composing preprocessors. Let $p_1,\ldots,p_n$ be preprocessors for which p_i^+ is well defined, and π a permutation of $\{1,\ldots,n\}$. Then $\langle p_1,\ldots,p_n\rangle_\pi$ is defined to be $(p_{\pi 1}^+;\ldots;p_{\pi n}^+)^+$. This is well defined since each p_i has a well-founded ordering $\prec_{p_i}$ and so their lexicographical ordering is a well-founded ordering for $\langle p_1,\ldots,p_n\rangle_\pi$. For example, for $n=3$, for p_i being a_i, and for π being $(2,3,1)$ this yields the preprocessor $(a_2^+;a_3^+;a_1^+)^+$.

3 Database of games

We can leverage the algebra for preprocessors to a query language over a set of games.

Query language. The query language is a fragment of first-order logic where formulae are closed and contain only a single and top-most quantification. The grammar for queries is given by

$$q ::= \forall \mathsf{G}\colon b \mid \exists \mathsf{G}\colon b \tag{2}$$

where G is a *fixed* variable that ranges over all games in a specified set of games $\mathscr{D}$ (a database), and b is the yet unspecified body that can only mention variable G, which binds to games G. The grammar for b is extensible. For now, we will freely use relational and functional symbols within b in examples.

Figure 2 depicts examples of queries. Query (3) asks whether there is a game in the database that is resilient to preprocessor p, since the equality $G = \mathsf{res}(G,p)$ means that p cannot simplify *anything* in game G. If p happens to be a very powerful preprocessor, a witness game G for the truth of this query may then be a good benchmark for solvers.

Query (4) asks whether preprocessors p and q have the same effect on all games of the database. If so, this does of course not necessarily imply that they have the same effect in general. This pattern has many uses, we mention two: Firstly, for q being $p;p$, e.g., we can test whether p is idempotent on games from our database. Secondly, if q is an optimization of p, we can test whether this optimization is correct for games in our database (a form of regression testing).

For an example of the second kind, let p be $a;\mathsf{lft}(a)$ and q be $\mathsf{lft}(a)$. Query (4) then tests on our database whether lft might be monotone in that it also does all the simplifications done by its argument a. This is not generally true as $\mathsf{lft}(a)$ only uses a conditionally, to probe whether certain nodes are won by certain players; it does not use a directly on the input game.

In query (5), $\mathsf{Sol}(G,p,0)$ denotes those nodes, if any, that preprocessor p classifies as being won by player 0 in game G. If p is a solver or a preprocessor that does decide the winners of some nodes, this query therefore checks whether p is correctly implemented (relative to the trusted implementation of some solver).

Query semantics. We explain the semantics of query evaluation informally. A model is a database $\mathscr{D}$. Evaluating query $\exists \mathsf{G}\colon b$ on $\mathscr{D}$ either returns an empty list (saying that no game satisfying b is in the database) or returns a game G from $\mathscr{D}$ satisfying b. Dually, the evaluation of $\forall \mathsf{G}\colon b$ either returns the empty list (saying that all games in the database satisfy b) or a game G from $\mathscr{D}$ that does not satisfy b. Both of these evaluations require the evaluation of b on a game G in $\mathscr{D}$, returning a Boolean truth value. That evaluation uses the interpretations of relational and

$$\exists \mathsf{G}\colon \mathsf{G} = \mathsf{res}(\mathsf{G}, p) \tag{3}$$
$$\forall \mathsf{G}\colon \mathsf{res}(\mathsf{G}, p) = \mathsf{res}(\mathsf{G}, q) \tag{4}$$
$$\forall \mathsf{G}\colon \mathsf{Sol}(\mathsf{G}, p, 0) \subseteq \mathsf{Sol}(\mathsf{G}, \mathsf{TrustedSolver}, 0) \tag{5}$$

Figure 2: Example query patterns, instantiable with preprocessors p and q.

functional symbols in b and the standard semantics of propositional logic to determine whether G satisfies b. In particular, we interpret equality $G_1 = G_2$ between games as extensional identity: both games have the same game graph and colors of nodes.

Example 1 *Let* $\mathit{index}(G)$ *evaluate to the index of game* G*. A game* G *satisfies* $(\mathit{index}(G) > 5) \wedge \neg(G = \mathsf{res}(G, a_1^+))$ *iff* G *has index greater than* 5 *and is not resilient to the preprocessor* a_1^+*. Similarly, query* $\exists \mathsf{G}\colon (\mathit{index}(\mathsf{G}) > 5) \wedge \neg(\mathsf{G} = \mathsf{res}(\mathsf{G}, a_1^+))$ *might return the game in Figure 1 from our database; its index is* 6 *and it can be simplified by* a_1^+ *as already discussed.*

Our implementation has explicit mechnisms for controlling the choice of database for the evaluation of queries. We do not show these mechanisms in this paper for sake of brevity.

4 Implementation

Our prototype workbench is a fusion of a parity game solver component, a very scalable online storage facility for parity games supporting simple interfaces, a client component which talks to data servers, and a query component for analyzing results stored on the servers or derived from further computations performed on games.

Software architecture. The distributed and highly extensible architecture of our platform for query execution is shown in Figure 3. It is comprised of three parts:

- At its center, directly interfacing with users, the query server actively maintains registration of game servers and query execution processes, manages parsing and interpreting user queries at runtime, and merges computation results after query executions return from the query execution group.
- On the right, we have a collection of game servers. Each game server stores a particular class of parity games and other computed results related to each game instance. The game server provides a uniform interface for access of generic information. By default, queries will be directed to computations about games in all game servers. But users can specify in the query to only look at results from a particular game server. Users can also easily inspect data recorded in each server through a web browser.
- The query execution group contains a collection of parallel processes responsible for processing a submitted query. The query execution group, implemented using JGroup, is self balanced. When a process node is shut down for whatever reason or when a new process node becomes available in the group, the group will rebalance itself evenly throughout.

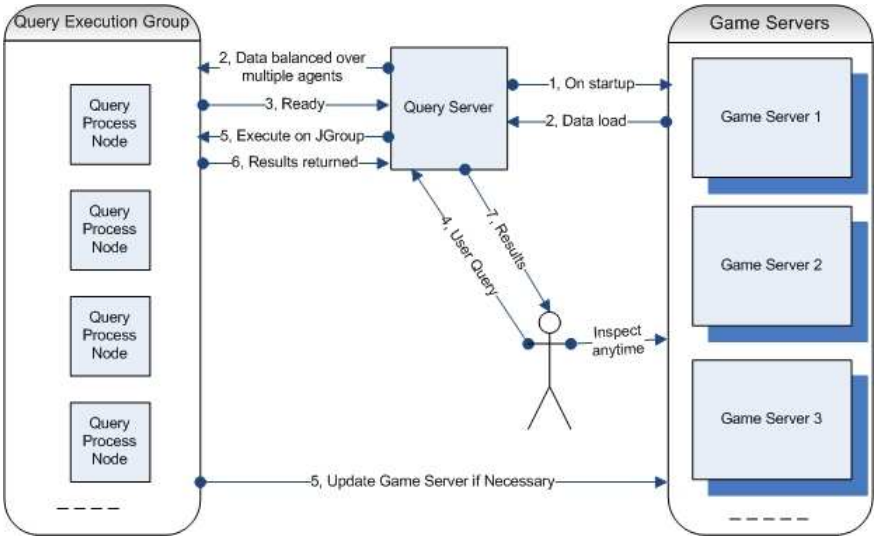


Figure 3: Overall architecture of the query engine for our workbench. And the typical sequence of interactions between user and tool.

Therefore, process nodes in the group could, in principle, reside on different machines and so facilitate parallelized query execution.

User session. Figure 4 shows a typical user session in our workbench. This session takes place on the query server, by that time all the games are already loaded into the process nodes. The user first enters a query as specified by the implemented query language, the server will then send the interpreted query to all query process nodes for local processing. Local results will be merged back at the query server. If the result is positive for a universally quantified query, the server will just return `true` and no witness is provided; if a contradiction is found, the server will return `false` and the associated witness. An existentially quantified query will return `true` and the witness if a positive example is found, and returns only `false` otherwise. The witness will be shown in both the `dot` description format and as a graph.

In this particular session the query asked whether, for all games, all nodes that are deleted by `A1;A2;A3;  P(A1;A2;A3)` are also deleted by `A2;A3;  P(A2;A3)`. This is not true, and the witness produced is displayed. (The meaning of the lifting `P` will be explained below.)

Implemented algebra. The following preprocessors are implemented in our prototype workbench: `A1` implements a_1^+, `A2` implements a_2^+, and `A3` implements a_3^+. Preprocessor composition $p;q$ is implemented as first running p on G and then running q on $\mathsf{res}(G,p)$. The iteration p^+ is implemented as a repeat-statement that initially runs p on G and then keeps executing p on the resulting game until a fixed point is reached.

Two types of functions are currently implemented. Function `L` is a more complex version of function lft described already on page 143 (we refrain from sketching the details here). Function `P` is a similar, less efficient transformation of predecessors that is based on our existing work in [ACH09, Wan07]. In the sequel, we write `PROBE[n](a)` for the preprocessor that initially runs `a`, then runs `P(a)`, etc., and stops after it has run the n-th nesting of `P` on `a`. For example,

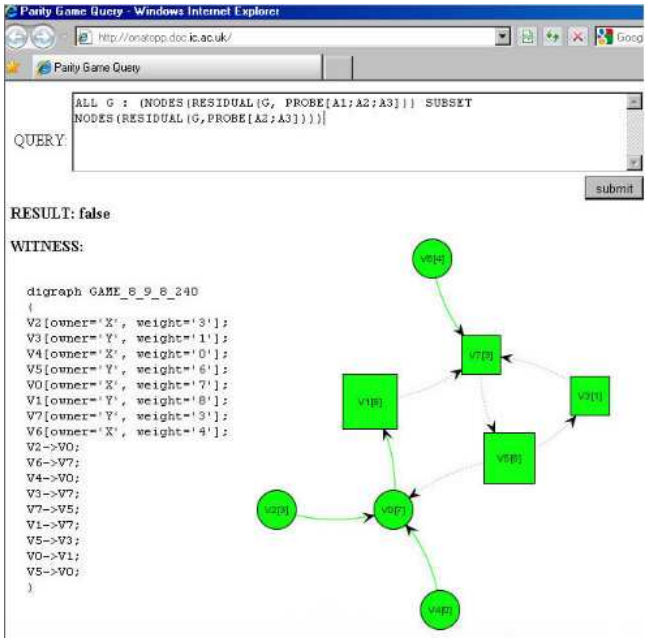


Figure 4: A typical user session on the query server, (1) the user enters a query; (2) the database of games is searched for a witness; (3) the interface then displays a game that refutes a universally quantified query or verifies an existentially quantified query (if applicable).

`PROBE[2](A1;A2;A3)` expands in this manner to the term

$$((\texttt{A1};\texttt{A2};\texttt{A3};\texttt{P}(\texttt{A1};\texttt{A2};\texttt{A3});\texttt{P}(\texttt{P}(\texttt{A1};\texttt{A2};\texttt{A3}))$$

Implemented query language. The query language we currently support is seen in Figure 5. There are three groups in the query language. The first group specifies that formulae have a single quantification and a body built from an adequate set of propositional connectives: `NOT`, `AND`, `OR`. The second group lists supported predicates, that allow reasoning about the game for its solutions, nodes, edges, strategies, and colors. For example, `SOLUTION(G, A, X)` returns those nodes of the game that preprocessor `A` decides to be won by player `X` – our keyboard encoding for player 0; player 1 is encoded by `Y`. The third group specifies preprocessors and their functions, here `L` and `P`. Apart from the three aforementioned preprocessors, this supports `EXP` which implements a solver based on Zielonka's algorithm [Zie98]. We give two further examples of how to write queries in this language:

```
ALL G : (NODES(RESIDUAL(G, L(A1;A2;A3))) SUBSET NODES(RESIDUAL(G, L(A2;A3))))
```

stipulates that all nodes, in all games, that are solved by `L` with preprocessor `A1;A2` are also solved by the same lifting function with preprocessor `A1;A2;A3`. Query

```
ALL G : (NODES(RESIDUAL(G, C(L(L(A2;A3))))) SUBSET NODES(RESIDUAL(G, EXP)))
```

states that all games are fully solved by iterating two nestings of `L` when applied to preprocessor `A2;A3`. This is so since $\mathsf{res}(G, EXP)$ is the "empty" game for *any* complete solver `EXP`.

```
QueryLanguage := QueryType GVar : Constraints
QueryType := ALL | SOME
Constraints := (Constraint) | (NOT Constraints) |
               (Constraints AND Constraints) | (Constraints OR Constraints}

Constraint := Fragment == Fragment | Number >= Number | Number <= Number |
              Number > Number | Number < Number | Nodes SUBSET Nodes |
              Edges SUBSET Edges | Colors SUBSET Colors
Fragment := Game | Nodes | Edges | Number | Colors
Game := GVar | RESIDUAL(Game, Prep)
Nodes := SOLUTION(Game, Prep, Player) | NODES(Game)
Edges := EDGES(Game)
Number := COUNT(Nodes) | COUNT(Edges) | COUNT(Colors)
Colors := COLORS(Game)
Player := X | Y

Prep := Atom | Prep; Prep | L(Prep) | P(Prep)
Atom := A1 | A2 | A3 | EXP
```

Figure 5: Implemented query language of our prototype workbench.

5 Discussion

We discuss some implementation issues that have relevance to the overall aims of the workbench.

Data model. An essential type of object on the game server is a *scratchpad*, given in the form of key/value pair. Resources, another essential object of our implementation, may be associated with multiple scratchpads. For example, for a scratchpad associated with a game a key may be the name of the game and the value the description of the game. Games may also have associated scratchpads that record solutions, solver statistics, etc. The system is agnostic of how scratchpads are being manipulated. Information submitted and accessed through a registered user account on the system is therefore interpreted by users or agents at the client side. This data model allows our workbench to be smoothly extended to work with other types of games, e.g. stochastic parity games [CJH04], with similar work flow requirements.

Populating databases. At present, we populate databases with randomly generated games and precompute and store the effects of many basic compositions of preprocessors. Although the reliance on random games does have inherent limitations, our workbench supports the specification and storage of any kind of game, e.g. known worst-case examples for specific solvers. Non-random games need to be entered manually and so we can expect to only support a limited database of such games.

At present, the generation of random parity game data takes place outside of the workbench, via a command line Java executable. After a game is generated, it is automatically populated to the specified server. Several other processes, also invoked from the command line, pick up games from the game server and prepare and attach solutions on to the game servers.

We now describe how we generate random games. For a game G with $|G| = n$, the index of V_G can be at most n, and $|E_G|$ ranges from n to $n \cdot (n-1)$ – since we have no self-loops and no deadlocks. For each possible value i of $|E_G|$ in that range, we generate $(i \cdot n)^2$ different games at random. A random seed is selected. For each possible value of i, the owners of the n nodes

as well as their colors are decided based on random sequences generated from that random seed. Such a sequence is also used to decide which edges should be present, ingoring self-loops and avoiding deadlocks until i edges are found. In this manner, we generated $100,352$ random games with 8 nodes each.

Comparing preprocessors or solvers. Probably the most involved analyses are performance comparisons between solvers, as head-to-head comparisons on implemented solvers. Such comparisons may be tainted by implementations that optimize data structures for specific solvers. Therefore, the solver package offered in this platform decouples the data structure and the algorithms, by having the latter work on an abstract parity-game interface. Researchers can build their solver algorithm for this interface and use the default parity game data structure implementation provided to compare against other implemented algorithms running on the same data structure. Alternatively, different data structures can be used to implement the same interface and one target algorithm can be tested for performance when applied to different data structures.

A more straightforward analysis is a scalability analysis where the interest is merely in finding out how an algorithm implemented in another language and context performs over a very large data set or on very large games. This is achieved by using the connector client to download the parity game data from the data servers in batches and to solve the games locally. Run-time statistics can then be compared to results from other solvers that ran on the same platform.

Parser and query optimization. The query parser and processor are rapid prototypes written in Java. There are many issues with this choice.

- It currently does not share common sub-expressions and so the meaning of such shared expressions is re-computed for each game.
- It is difficult to define pattern matching rules and query optimization paths in Java. This could be easily implemented in Prolog.
- Prolog would also allow us to guide search, so that less expensive sub-expressions get evaluated first and so expensive sub-expressions may not have to be evaluated, as in a conjunction `EXPENSIVE AND CHEAP`.
- The current version of the query language only supports a very limited set of operations because they are cumbersome to implement correctly in Java. For example, we may want to query for a game with a node having n outgoing edges to nodes owned by its opponent. Prolog would make it much easier to build such queries.
- A potential problem with migrating parsing and executing queries from Java to Prolog is the need of call-backs from the Prolog to the Java process. We are currently evaluating the performance impact of such a need.

Memory footprint. Because all data about all parity games are loaded into the memory in uncompressed format, the combined required memory for all process nodes can be huge. Assuming each game only occupies 10KB in memory, a dataset of 10 million games requires around 100GB of memory footprint. The ability to distribute process nodes over machines will help, but won't achieve scalability in and of itself. Two additional solutions suggest themselves. Firstly, we might store Boolean matrices that record values of atomic query expressions for games. The complete witness information could then be recomputed for the chosen witness. Secondly, we

might generate games on a hierarchy of game server arrays that would act like a sieve so that games pass through to higher level servers only if they "survive" specified queries. Initial experiments suggest that this can eliminate at least 95 percent of randomly generated games.

6 Using the workbench

We now illustrate how one can use the current workbench prototype to evaluate and validate preprocessors and solvers. In doing so, we also generate some games that may serve as a first generation of benchmarks for solvers. Figure 6 shows four interesting games, found on a database populated with more than 100,000 random 8-node games.

Witness (A) is fully solved by `PROBE[0](A2;A3)` but not by `PROBE[2](A3)`. That is to say, the game is fully solved by `A2;A3` but not by `A3; P(A3); P(P(A3))`. This is perhaps surprising since the latter incrementally nests a lifting function whereas the former does not lift at all. But the latter uses a slightly weaker preprocessor and this weakness is not being compensated for in this witness game.

Witness (B) is solved by `A2;A3` but not by `A1;A2;A3`. This seems counter-intuitive since the initial application of a color reduction preprocessor appears to harm the effectiveness of subsequent preprocessing. But `A1` may close some "color gaps" in the game and those very gaps may enable `A2` to reduce some color to 0.

Witness (C) is solved by `A2;A3` or by `L(A2;A3)` but not by `L(L(A2;A3))`. This means that the non-idempotent lifting function `L` is not always more powerful than its previous nesting version. Witness (C) can in fact not be solved by any further nestings of `L` applied to `A2;A3` (we refrain from sketching the argument here). Applying the iteration operator `C` to each function call of `L` would make higher nestings more powerful than lower ones.

Finally, witness (D) shows an 8-node game that is resilient to `PROBE[3](A3)`, i.e. the preprocessor leaves the game unchanged.

We also experimented with generating datasets for 64-node and 128-node games. Fig. 7 shows a 64-node, 320-edge game whose subgame of grey nodes is resilient to `PROBE[5](A2;A3)`. This residual game is therefore resilient to the application of `P` to `A2;A3`, for any nesting up to level 5, suggesting it is reasonably complex to solve in general.

7 Future work and conclusions

In future work, we mean to address the identified implementation issues and extend the query language with some features that our use of the tool revealed as being desired. The identification of further preprocessors and their implementation are also planned. In the medium term, we mean to implement plain-vanilla versions of the most prominent solvers so that we can begin with evaluating them on generated benchmarks. Hopefully this will allow us to assess the utility of specific preprocessors for generating benchmarks. We also mean to create a database that stores known worst-case games for specific solvers. We also mean to determine whether the workbench can be used to design and immediately test novel solvers. Our workbench currently represents games explicitly. We mean to investigate how symbolic representations of games can be incorporated so that symbolic algorithms can be supported as well.

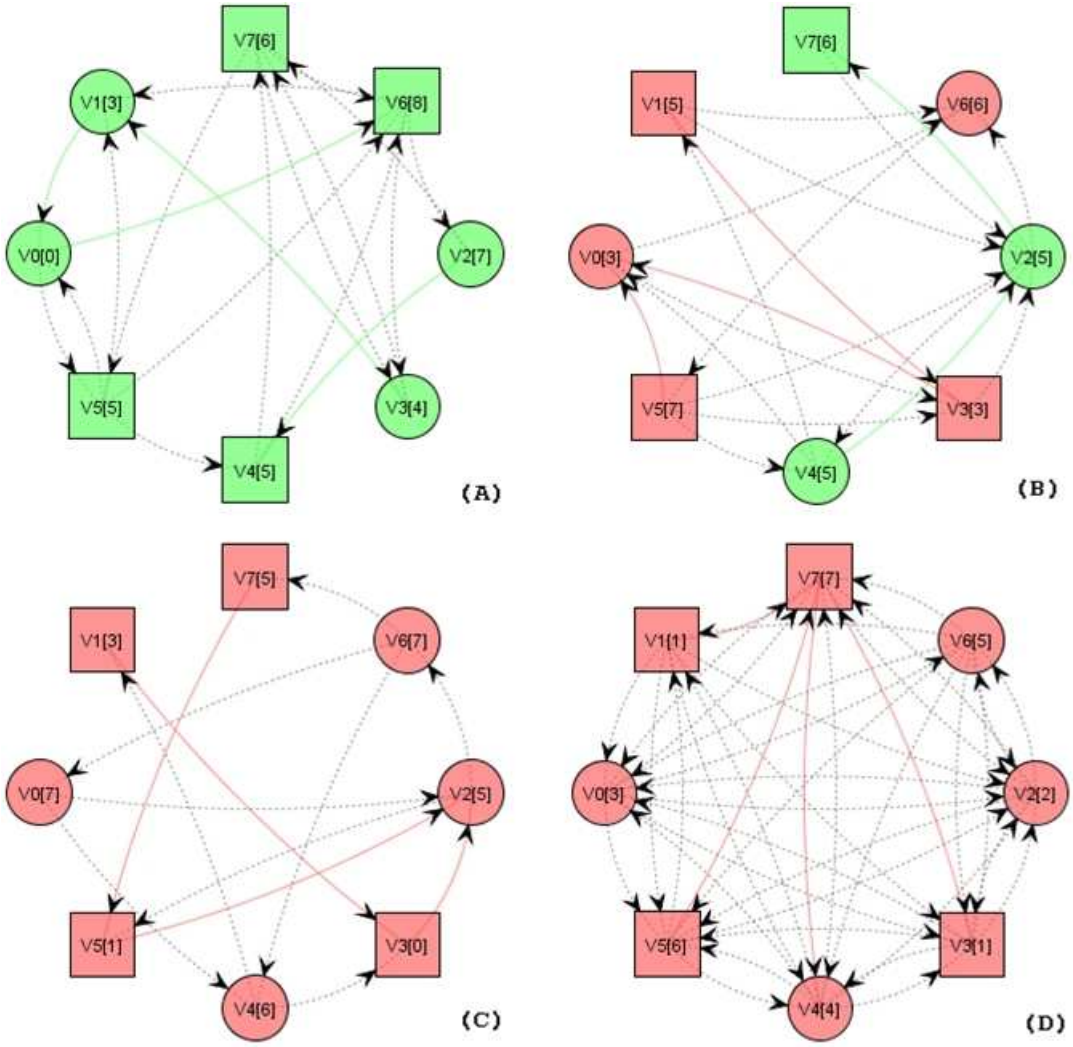


Figure 6: Specific 8-node games found using the query server. Witness (A) is solved by `PROBE[0](A2;A3)` but not by `PROBE[2](A3)`; witness (B) is solved by `A2;A3` but not by `A1;A2;A3`; witness (C) is solved by `A2;A3` or by `L(A;2A3)` but not by `L(L(A2;A3))`; witness (D) is resilient to `PROBE[3](A3)`.

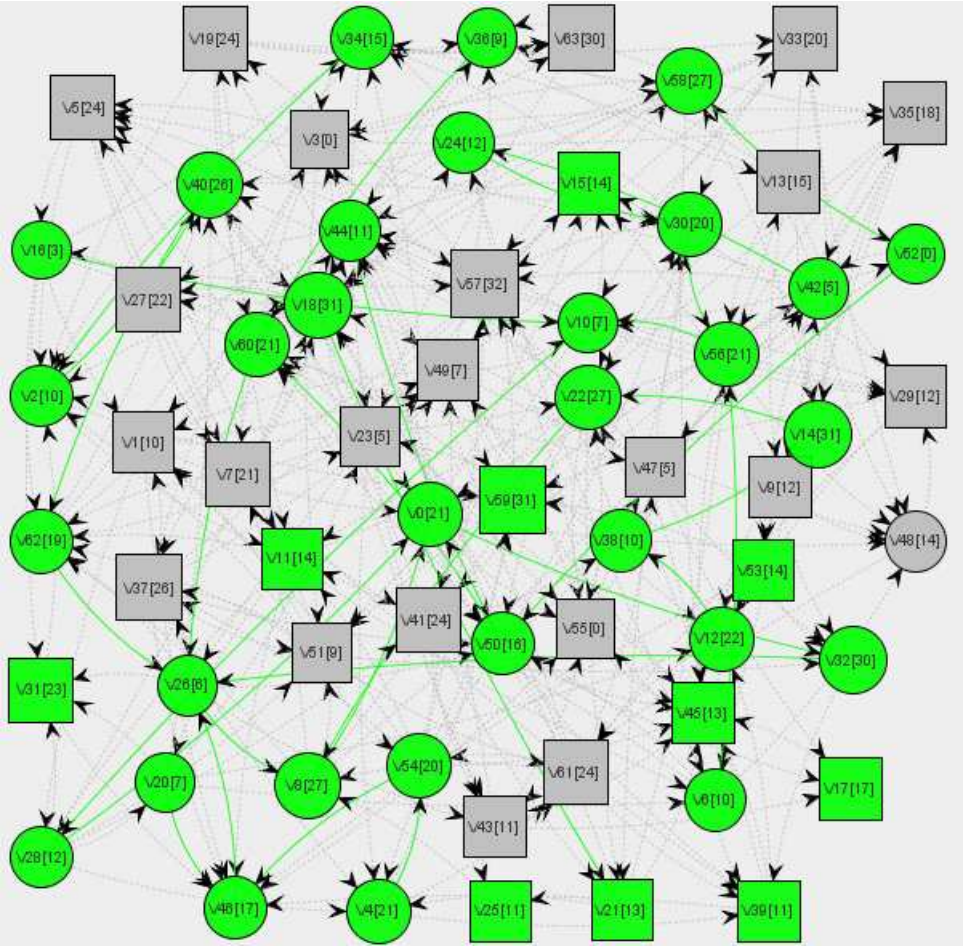


Figure 7: A 64-node game with 320 edges. Its sub-game of grey nodes is resilient to `PROBE[5](A2;A3)`.

At the more theoretical end, we mean to investigate whether our query language is presentable in a logic for which the synthesis problem is known to be decidable (this may not be straightforward due to the presence of counting primitives). The hardness of games under specific preprocessors may also be related to the descriptive complexities of such games. Such connections between the expressiveness of fixed-point logics and descriptive set theory have been identified, e.g. in [Bra03].

We summarize. In this paper we argued the utility of a workbench that can generate and store parity games with two ends in mind: to aid in the design, validation, and evaluation of preprocessors for parity game solvers; and to aid in the generation of benchmark parity games that are meaningful for a wide range of solvers. We sketched a framework that supports easy composition of preprocessors, offers a query language on a database of games for generating games of interest, and supports scalable query evaluation. A prototype implementation of this framework has been described and example interactions with that workbench were provided to demonstrate its potential in relation to the aforementioned two ends.

Acknowledgements: This research was, in part, supported by the UK EPSRC project *"Complete and Efficient Checks for Branching-Time Abstractions"(EP/E028985/1)*.

Bibliography

[ACH09] A. Antonik, N. Charlton, M. Huth. Polynomial-Time Under-Approximation of Winning Regions in Parity Games. *Electronic Notes in Theoretical Computer Science* 225:115–139, January 2009.

[Bra03] J. C. Bradfield. Fixpoints, games and the difference hierarchy. *ITA* 37(1):1–15, 2003.

[CJH04] K. Chatterjee, M. Jurdzinski, T. A. Henzinger. Quantitative stochastic parity games. In *Proc. of SODA'04*. Pp. 121–130. 2004.

[FL09] O. Friedmann, M. Lange. Solving Parity Games in Practice. In *Proc. of ATVA'09*. Springer, 2009. to appear.

[JPZ08] M. Jurdzinski, M. Paterson, U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. *SIAM J. Comput.* 38(4):1519–1532, 2008.

[Jur98] M. Jurdzinski. Deciding the Winner in Parity Games is in UP ∩ co-UP. *Inf. Process. Lett.* 68(3):119–124, 1998.

[MS95] D. E. Muller, P. E. Schupp. Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of the Theorems of Rabin, McNaughton and Safra. *Theor. Comput. Sci.* 141(1&2):69–107, 1995.

[PR89] A. Pnueli, R. Rosner. On the Synthesis of a Reactive Module. In *Proc. of POPL'89*. 1989.

[Sti95] C. Stirling. Lokal Model Checking Games. In *Proc. of CONCUR'95*. Pp. 1–11. Springer, 1995.

[Wan07] H. Wang. Framework for Under-Approximating Solutions of Parity Games in Polynomial Time. Master's thesis, Department of Computing, Imperial College London, June 2007.

[Zie98] W. Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theor. Comput. Sci.* 200(1-2):135–183, 1998.

Faster FDR Counterexample Generation Using SAT-Solving

H. Palikareva, J. Ouaknine and A. W. Roscoe

Oxford University Computing Laboratory, Oxford, UK

Abstract: With the flourishing development of efficient SAT-solvers, bounded model checking (BMC) has proven to be an extremely powerful symbolic model checking technique. In this paper, we address the problem of applying BMC to concurrent systems involving the interaction of multiple processes running in parallel. We adapt the BMC framework to the context of CSP and FDR yielding bounded refinement checking. Refinement checking reduces to checking for reverse containment of possible behaviours. Therefore, we exploit the SAT-solver to decide bounded language inclusion as opposed to bounded reachability of error states, as in most existing model checkers. We focus on the CSP traces model which is sufficient for verifying safety properties. We present a Boolean encoding of CSP processes resting on FDR's hybrid two-level approach for calculating the operational semantics using supercombinators. We describe our bounded refinement-checking algorithm which is based on watchdog transformations and incremental SAT-solving. We have implemented a tool, SymFDR, written in C++ which uses FDR as a shared library for manipulating CSP processes and the state-of-the-art SAT-solver MiniSAT. Experiments indicate that in some cases, especially for complex combinatorial problems, SymFDR significantly outperforms FDR.

Keywords: CSP, FDR, concurrency, process algebra, Bounded Model Checking, SAT-solving, safety properties

1 Introduction

Model checking techniques can be partitioned into those which are *symbolic*, based on abstract representation of sets of states, and those which are based on *explicit* examination of individual states. The former generally represent sets of states as formulae in Boolean logic and use techniques such as SAT-solving and BDD manipulation to decide checks. The latter can be enhanced by techniques such as hierarchical state-space compression and partial-order methods. The main obstacle when applying these approaches in practice is the *state-space explosion problem* by which the number of states in a system grows exponentially with the number of parallel components and also the number and bit sizes of data values.

FDR [Ros94, G+05] is a long-established tool for the refinement checking of CSP [Hoa85, Ros98]. When deciding whether a proposed implementation process *Impl* refines a normalised specification process *Spec*, FDR follows algorithms exploring the Cartesian product of the state spaces of *Spec* and *Impl* in a way comparable to conventional model checking. Therefore, until now, FDR has followed the explicit model checking approach. There has been, however, some work on the symbolic model checking of CSP [PY96, SLDS08].

This paper reports our attempts to integrate SAT-based bounded model checking [BCCZ99] into FDR. We show how the same internal structures used in FDR's two-level representation of state spaces can be translated readily into Boolean logic. Within the scope of this paper, we only consider the translation of *trace* refinement to SAT checking.

The result is a prototype tool SymFDR which, when combined with state-of-the-art SAT-solvers such as MiniSAT [ES03a, EB05], sometimes outperforms FDR by a significant margin when finding counterexamples. We compare the performance of SymFDR with the performance of FDR, FDR used in a non-standard way, PAT [SLD08] and, in some cases, NuSMV [CCG+02], Alloy Analyzer [Jac06] and straight SAT encodings of the problems under consideration.

The remainder of the paper is organised as follows. In Section 2, we set out the necessary background on CSP and FDR's two-level strategy for performing refinement checks. We briefly describe the ideas underlying BMC. In Section 3, we show how to adapt the watchdog approach [RGM+03] to BMC, while in Section 4, we summarise the methods we use to translate FDR's supercombinator representation of a state machine into input for a SAT-solver. Section 5 gives details of how SymFDR is built on top of this, and Section 6 offers experimental comparisons.

2 Preliminaries

2.1 CSP and FDR

In this section, we assume that the reader is familiar with CSP and we therefore give only a brief overview of CSP and FDR. The interested reader is referred to [Ros98]. Furthermore, we restrict our focus exclusively to the traces model, intentionally omitting information about other more expressive models of CSP.

2.1.1 CSP Syntax

In this section, we recall the core syntax of CSP. Let Σ be a finite alphabet of (visible) events with $\tau, \checkmark \notin \Sigma$. The internal action τ occurs silently and is invisible outside a process. $\checkmark$ denotes a successful termination of a process. In what follows, we assume that $a \in \Sigma$, $A \subseteq \Sigma$ and $B \subseteq \Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$. $R \subseteq \Sigma \times \Sigma$ denotes a renaming relation on Σ.

Definition 1 A CSP process is defined recursively via the following grammar:

$$\begin{aligned} P := \quad & STOP \mid SKIP \mid DIV \mid x : A \rightarrow P(x) \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \mid \\ & P_1 \underset{B}{||} P_2 \mid P_1 ; P_2 \mid P \setminus A \mid P[\![R]\!] \mid \mu P \bullet F(P) \end{aligned}$$

CSP_M converts core CSP into an ASCII form and adds several further operators and an extensive functional language. SymFDR supports the full CSP_M syntax, except that it cannot at present handle scripts using the function *chase*.

2.1.2 Denotational Semantics

CSP supports a hierarchy of several denotational semantic models. Each of them describes a process in terms of the observable behaviours it can exhibit. All denotational models are

compositional in the sense that the denotational value of each process can be computed in terms of the denotational values of its subcomponents.

In the traces model, a process P is identified with the set of its finite traces, denoted by $traces(P)$. Intuitively, a trace of a process is a sequence of visible actions that the process can perform. The set of traces of a process is non-empty and prefix-closed.

There are two different approaches for obtaining the set $traces(P)$ — either by constructing it inductively from the traces of its subcomponents, or by extracting it from the operational semantics. Refer to [Ros98] for the rules underlying the first approach. Since denotational values of processes are rather complex and often infinite, FDR calculates the behaviours of a process from its standard operational representation which is justified by semantic models being congruent to it. The congruence theorems are presented and proven in [Ros98].

2.1.3 Operational Semantics

The operational semantics models CSP processes as labelled transition systems (LTS's), with nodes denoting processes and labels denoting visible or τ actions. Since the LTS representation is not unique, in terms of the operational semantics, two processes are considered equivalent if they are strongly bisimilar [Ros98]. The operational semantics is calculated by repeatedly applying a set of inference rules, called *firing rules*. Firing rules provide recipes for constructing an LTS out of a CSP description of a process. The recipes define how processes can evolve by calculating the initial actions available at each node and the possible results after performing each action. The reader is referred to [Ros98] for more information.

Extracting Behaviours from Operational Semantics. We now present how behaviours, in our case – traces, can be retrieved from the operational semantics of a process.

Formally, a labelled transition system is a quadruple $M = \langle S, s_0, L, T \rangle$, where S is a finite set of states, $s_0 \in S$ is the initial state, L is a finite set of labels, $T \subseteq S \times L \times S$ is the transition relation.

For convenience, we write $s \xrightarrow{l} s'$ instead of $(s,l,s') \in T$. Furthermore, we write $s \xrightarrow{l}$ if there exists $s' \in S$, such that $s \xrightarrow{l} s'$. For $s \in S$ and $l \in L$, we define $Post(s,l) = \{s' \in S | s \xrightarrow{l} s'\}$ — the set of direct *l-successors* of s. M is then is deterministic if, for any $s \in S$ and $l \in L$, $|Post(s,l)| \leq 1$. An execution of M is a finite or an infinite alternating sequence of states and events $\pi = s_0 l_1 s_1 l_2 \ldots l_n s_n \ldots$ such that s_0 is the initial state and for all i, $s_i \xrightarrow{l_{i+1}} s_{i+1}$.

Let P be a finite-state process and $OS_P = \langle S^P, s_0^P, L^P = \Sigma^{\tau,\checkmark}, T^P \rangle$ be the LTS underlying the operational semantics of P. We denote by α_P the set of all visible events that P can perform, i.e. $\alpha_P = \Sigma^{\checkmark}$. We write $\Sigma^{*\checkmark}$ to denote the set of finite words over Σ which might end with $\checkmark$, and similarly, $(\Sigma^{\tau})^{*\checkmark}$. For $p,q \in S^P$, we use the following notation:

- $initials(p) = \{l \in \Sigma^{\checkmark} | p \xrightarrow{l}\}$, i.e. $initials(p)$ is the set of visible events that can be communicated from the state p.
- for $t = \langle x_i | 0 \leq i < n \rangle \in (\Sigma^{\tau})^{*\checkmark}$, we write $p \xmapsto{t} q$ if there exists a sequence of states $p_0, p_1, \ldots, p_n$, such that $p_0 = p$, $p_n = q$ and $p_k \xrightarrow{x_k} p_{k+1}$ for $k \in \{0, \ldots, n-1\}$.
- for $t \in \Sigma^{*\checkmark}$, we write $p \Longrightarrow q$ if there exists $t' \in (\Sigma^{\tau})^{*\checkmark}$, such that $p \xmapsto{t'} q$ and $t = t' \upharpoonright \Sigma^{\checkmark}$, i.e. t is t' with all the τ's removed.

Then, we define $traces(P) = \{t \in \Sigma^{*\checkmark} | \exists q \in S^P . s_0^P \Longrightarrow q\}$.

The Two-Level Approach. In fact, FDR exploits a hybrid high-/low- level approach for calculating the operational semantics of a process [Ros08]. Generally, the low level comprises all true recursions, the high level – processes composed by parallel composition, hiding and renaming, although the dividing line is a bit more complex. For each process compiled on the low level, an explicit LTS is produced, following the firing rules. Compiling on the high-level is called *supercompiling*. It is based on calculating a set of rules for turning a combination of LTS's into a single LTS, without explicitly constructing it. For most practical examples, the result of supercompilation is a high-level structure.

The high-level structure consists of two parts. The first one is a process tree with leaves – low-level compiled LTS's, and internal nodes – CSP operators such as hiding, renaming or parallel composition. Each node, even if internal, represents a process and is interpreted as an LTS with its behaviours deducible from the behaviours of its children on-the-fly. The second part of high-level structure is a set of rules mapping actions of a number of leaf processes to an event-outcome of the composite root process [Ros98]. Those rules are called *supercombinators*. In what follows, we use the notions of supercombinators and rules interchangeably. Within a supercombinator, each process can participate with a visible event, a silent action τ, or not be involved at all. The supercompiler generates the following types of rules [Ros98, RRS+01]:

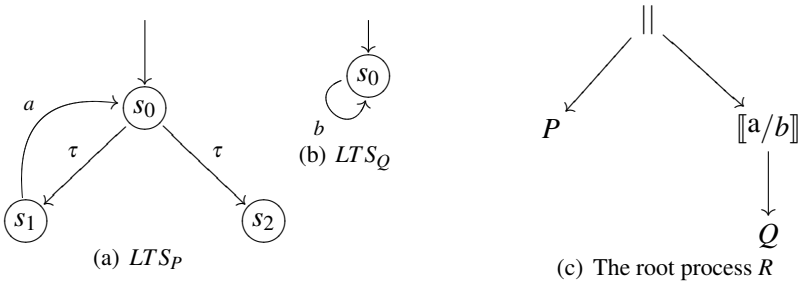
- a rule for a leaf willing to perform a τ which promotes a τ action of the root process
- rules using visible actions

Note that the visible actions that the leaf processes perform need not be the same if hiding or renaming is involved in the combination being modelled. For example, if $P = a \rightarrow P$ and $Q = b \rightarrow Q$, then if P performs a and Q performs b, $P \underset{\{a\}}{||} Q[\![a/b]\!]$ can perform a, where $Q[\![a/b]\!]$ is the process Q with the event b being renamed to a. Hence, (a,b,a) is a valid rule for the root process $P \underset{\{a\}}{||} Q[\![a/b]\!]$ with leaves P and Q: the first two elements of the rule triple represent the actions of P and Q, respectively, the last element provides the event-outcome of $P \underset{\{a\}}{||} Q[\![a/b]\!]$.

In FDR, at every particular step the leaf processes can be either *switched on* or *switched off*. Processes are switched on if they are currently under a CSP operator that makes their actions immediately relevant for the action of the overall system. Processes are switched off if they are under a CSP operator that does not need their actions to deduce the resulting action of the system. For instance, in $P_1|||P_2$, both P_1 and P_2 are switched on. In $P_1;P_2$, P_2 is switched off until P_1 performs $\checkmark$. In $a \rightarrow (P_1|||P_2)$, both P_1 and P_2 are initially switched off until a is communicated.

We refer to the different configurations of switched on and switched off leaf processes as *formats*. In the worst case, there could be exponentially many different formats, but in practice this is rarely the case. In FDR, the set of supercombinators is partitioned with respect to the existing formats. Hence, supercombinators can be viewed as dynamic or static, depending on whether they switch the system to another format after being triggered or not.

A state of the root high-level process, also called a *configuration*, is a tuple of the current states of its leaf processes. When running the root process, FDR computes its initial actions by checking which supercombinators are enabled from the current configuration and the current format of the root. A supercombinator might be disabled if not all leaf processes are able to communicate the event they are responsible for within the supercombinator. Hence, the operational semantics



(a) LTS_P (b) LTS_Q (c) The root process R

Figure 1: Example OS

of the root can be considered an implicit LTS, whose transitions can be switched on and off. The states are represented by a pair of a configuration and a format of the root. Transitions are modelled by supercombinators. We formalise these notions when describing our Boolean encoding of CSP processes. In this section, we illustrate the two-level approach with a small example.

Example 1 *Let us consider the process* $R = P \underset{\{a\}}{||} Q[\![a/b]\!]$, *where* $P = a \rightarrow P \sqcap STOP$ *and* $Q = b \rightarrow Q$. *The process tree of R is presented on Figure 2(c). The explicit LTS machines underlying the semantics of the leaves P and Q are depicted on Figure 2(a) and Figure 2(b), respectively. The root process R contains a single format with two rules — one rule stating that if P performs* τ *then the entire system performs* τ *and another rule stating that if P performs a and Q performs b, R can perform a.*

2.1.4 Refinement Checking

Given two CSP processes *Spec* and *Impl*, the refinement check $Spec \sqsubseteq Impl$ reduces to checking for reverse containment of possible behaviours. For the traces model, $Spec \sqsubseteq_T Impl$ iff $traces(Impl) \subseteq traces(Spec)$.

We briefly outline how FDR carries out the refinement check. Let $OS_{Spec} = \langle S^s, s_0^s, L^s, T^s \rangle$ and $OS_{Impl} = \langle S^i, s_0^i, L^i, T^i \rangle$ be the labelled transition systems representing the operational semantics of *Spec* and *Impl*, respectively. As a preprocessing step, FDR normalises OS_{Spec}, so that OS_{Spec} reaches a unique state after any trace. The normalisation procedure requires as a precondition that OS_{Spec} be explicated and therefore *Spec* sequentialised. Essentially, the normalisation procedure transforms OS_{Spec} into the unique equivalent τ-free deterministic LTS with the fewest possible states (bisimulation-reduced). After normalising OS_{Spec}, FDR traverses the cross-product of OS_{Spec} and OS_{Impl} in a breadth-first manner, checking for compatibility of mutually-reachable states. For the traces model, a pair of states (s^s, s^i) is compatible, if $initials(s^i) \subseteq initials(s^s)$.

2.2 Bounded Model Checking

SAT-based bounded model checking [BCCZ99] is a symbolic model checking technique considered complementary to BDD-based model checking [BCM+92, McM93].

Bounded model checking focuses on searching for counterexamples of bounded length only.

The underlying idea is to fix a bound k and to unwind the model for k steps, thus considering behaviours and counterexamples of length at most k. In practice, BMC is conducted iteratively by progressively increasing k until either a counterexample is detected or k reaches a precomputed threshold called *completeness threshold* [CKOS05], which indicates that the model satisfies the specification. It is important to note that without knowing the completeness threshold, the BMC procedure is incomplete. Hence, BMC is mostly suitable for detecting bugs, not for verification (proving absence of bugs).

SAT-based BMC [BCCZ99] reduces the model checking problem to a propositional satisfiability problem. The idea is to construct a Boolean formula which is satisfiable if and only if there is a counterexample of length k. This formula is fed into a SAT-solver which decides the model checking problem in question and produces a counterexample, if any. Due to the DFS-nature of the SAT decision procedure, this technique allows for a fast detection of counterexamples.

3 Performing Bounded Trace Refinement

In this section, we present our iterative bounded refinement checking algorithm. Our approach for establishing trace refinement is based on watchdog transformations [RGM+03]. Our objective is the following. We are given two CSP processes *Spec* and *Impl* and an integer k. We aim at checking whether $Spec \sqsubseteq_T^k Impl$, i.e., whether all executions of the implementation of length at most k agree with the specification.

3.1 Preprocessing Phase Using FDR

Our implementation is intended as an alternative back-end for FDR, orthogonal to the standard explicit method of performing trace refinement. Currently, we use a shared library version of FDR for manipulating CSP processes and we mimic FDR up to the point of the final state-space exploration phase. Therefore, SymFDR reuses FDR's compiler and supercompiler and the data structures underlying the operational semantics.

At present, we use FDR to supercompile and normalise *Spec* and to retrieve LTS_{Spec} representing the operational semantics of *Spec*.

We assume that the implementation *Impl* comprises the interaction of c sequential processes $P_1, \ldots, P_c$ running in parallel, possibly using hiding and renaming. We write $Impl = P_1||P_2 \ldots ||P_c$ to denote a high-level process *Impl* with leaf processes $P_1, \ldots, P_c$. We use FDR to supercompile *Impl* and to retrieve both the set of supercombinators and the set $\{LTS_{P_i} | i \in \{1, \ldots, c\}\}$.

3.2 Watchdog Refinement Checking Algorithm

In a nutshell, the main steps of our algorithm are the following:

1. We transform *Spec* into a process *Watchdog* which allows the behaviours of both *Spec* and *Impl*. The transformation adds a special state *sink* to LTS_{Spec} and forces all erroneous traces (traces that do not conform with *Spec*) to be directed to *sink*.
2. We construct a process $Refinement = Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{||} Impl =$ $Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{||} (P_1||P_2 \ldots ||P_c)$

3. We check whether *Watchdog* can reach its *sink* state within k steps of the execution of *Refinement*.

The Watchdog Process. The transformation we apply on *Spec* is performed at the level of LTS_{Spec}. We add a state *sink* and make LTS_{Spec} total with respect to the alphabet $\alpha_{Spec} \cup \alpha_{Impl}$. The resulting process *Watchdog* operationally passes through *sink* whenever executing a trace that is not allowed by *Spec*. We allow an execution of *Watchdog* to contain any number of τ's after visiting *sink* in order to be able to increase the BMC bound by more than 1 at each step.

The Refinement Process. The process $Refinement = Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{||} (P_1||P_2 \ldots ||P_c)$ can be used as an indicator whether *Impl* can behave in a way incompatible with *Spec*. *Watchdog* becomes just one of the sequential leaf processes of *Refinement*. It is evident then that:

1. $Spec \sqsubseteq_T Impl \Longleftrightarrow Watchdog$ never reaches its *sink* state in any execution of *Refinement*
2. All executions of *Refinement* forcing *Watchdog* to pass through its *sink* state constitute valid counterexamples of the assertion $Spec \sqsubseteq_T Impl$

4 Boolean Encoding of CSP Processes

In this section we present our encoding of CSP processes into Boolean formulae. First, we demonstrate how to encode sequential or explicated processes, corresponding to leaf processes in the operational representation. Then, we show how to glue together sequential processes with supercombinators to obtain an encoding of a high-level process. In what follows, we call a high-level process a concurrent system.

For the Boolean encoding we use the following notation. $\lceil X \rceil(Vars)$ denotes the Boolean encoding of X with respect to the vector(s) of Boolean variables *Vars*.

4.1 Encoding a Sequential Process

Let P be a finite-state process with alphabet of events Σ. Let $OS_P = \langle S, s_0, L = \Sigma^{\tau,\checkmark}, T \rangle$ be the LTS representing the operational semantics of P.

Encoding the Set of States. The basic idea is to enumerate the states in binary and represent them as Boolean functions. Each state $s \in S$ is identified by a bit-vector $\overline{b} = (b_1, \ldots, b_n)$ of size $n = \lceil \log_2 |S| \rceil$ using an injective encoding $enc_S : S \to \{0,1\}^n$. We introduce an ordered vector of n distinct Boolean variables $\overline{x} = (x_1, \ldots, x_n)$. Each variable x_i uniquely identifies its corresponding bit b_i and, for each $s \in S$, $\lceil s \rceil(\overline{x})|_{\overline{x}=\overline{b}} = 1$ iff $enc_S(s) = \overline{b}$. We define $\lceil I \rceil(\overline{x}) = \lceil s_0 \rceil(\overline{x})$.

Encoding the Set of Labels. Using the same technique, we introduce an ordered vector $\overline{y} = (y_1, \ldots, y_m)$ of $m = \lceil \log_2 |L| \rceil$ distinct Boolean variables for encoding the set of labels $L = \Sigma^{\tau,\checkmark}$.

Encoding the Transition Relation. In order to represent the transition relation T, we introduce a copy $\overline{x'} = (x'_1, \ldots, x'_n)$ of $\overline{x} = (x_1, \ldots, x_n)$. $\overline{x}$ serves for representing the source states of transitions, $\overline{x'}$ – for representing the destination states. Then, for $t = (s_{src}, l, s_{dest}) \in T$, $\lceil t \rceil(\overline{x}, \overline{y}, \overline{x'}) = \lceil s_{src} \rceil(\overline{x}) \wedge \lceil l \rceil(\overline{y}) \wedge \lceil s_{dest} \rceil(\overline{x'})$. For any $s \in S$, we write $\lceil s \rceil(\overline{x'})$ to denote $\lceil s \rceil(\overline{x})[\overline{x'} \leftarrow \overline{x}]$, i.e. we represent s with respect to the variables $\overline{x}$ and then substitute the variables $\overline{x}$ with $\overline{x'}$. The encoding of the entire transition relation is the following: $\lceil T \rceil(\overline{x}, \overline{y}, \overline{x'}) = \bigvee_{t \in T} \lceil t \rceil(\overline{x}, \overline{y}, \overline{x'})$.

Encoding Executions. We can now represent a sequential process P implicitly by the pair of functions $\langle \lceil T^P \rceil(\overline{x},\overline{y},\overline{x'}), \lceil I^P \rceil(\overline{x}) \rangle$. For a given integer k, we define $Paths(P,k)$ to be the set of all executions $s_0 l_1 s_1 l_2 \ldots l_k s_k$ of OS_P of length k. If flattened to traces, $Paths(P,k)$ might contain traces of P of size less than k if τ's are present in the executions. In order to represent $Paths(P,k)$ symbolically, we introduce $(k+1)$ vectors of n Boolean variables $\overline{x_0},\overline{x_1}\ldots\overline{x_k}$ and k vectors of m Boolean variables $\overline{y_1},\overline{y_2}\ldots\overline{y_k}$. The vectors $\overline{x_0},\overline{x_1},\ldots,\overline{x_k}$ represent the states $s_0,s_1,\ldots,s_k$, respectively. Likewise, the vectors $\overline{y_1},\overline{y_2},\ldots,\overline{y_k}$ represent the labels of the corresponding transitions. Then $\lceil Paths(P,k) \rceil(\overline{x_0},\overline{x_1}\ldots\overline{x_k},\overline{y_1},\overline{y_2}\ldots\overline{y_k}) = \lceil I^P \rceil(\overline{x_0}) \wedge \bigwedge_{i=0}^{k-1} \lceil T^P \rceil(\overline{x_i},\overline{y_{i+1}},\overline{x_{i+1}})$.

4.2 Encoding a Concurrent System

Since a high-level root process can be modelled as an LTS, we now show how to encode a concurrent system similarly to a low-level sequential process.

A concurrent system is a set of processes running in parallel possibly using renaming and hiding. We denote by $Sys(c)$ the interaction of c sequential processes $P_1,...,P_c$ communicating over sets of events $\Sigma_1,....,\Sigma_c$, respectively. Let $\Sigma = \cup_{i=1}^{c} \Sigma_i$, $m = \lceil \log_2 |\Sigma^{\tau,\checkmark}| \rceil$.

Encoding the Sequential Processes. For $i \in \{1,...,c\}$, let $OP^i = \langle S^i, s_0^i, L^i = \Sigma_i^{\tau,\checkmark}, T^i \rangle$ be the LTS representing the operational semantics of P_i. Since $\Sigma_i \subseteq \Sigma$, we actually consider $L^i = \Sigma^{\tau,\checkmark}$.

For each process P_i, let $n_i = \lceil \log_2 |S^i| \rceil$. In order to represent S^i and the transition relation T^i, we introduce two copies of n_i Boolean variables $\overline{x^i} = (x_1^i,...,x_{n_i}^i)$ and $\overline{x^{i'}} = (x_1^{i'},...,x_{n_i}^{i'})$. The construction of $\lceil T^i \rceil(\overline{x^i},\overline{y^i},\overline{x^{i'}})$ and $\lceil I^i \rceil(\overline{x^i})$ follows the ideas from Section 4.1.

As illustrated in Section 4.1, for each process P_i we introduce a vector of m Boolean variables $\overline{y^i} = (y_1^i,...,y_m^i)$ for encoding the set $L^i = \Sigma^{\tau,\checkmark}$ symbolically. Thus, each process has its own set of variables for representing the alphabet $\Sigma^{\tau,\checkmark}$. We introduce an additional vector of Boolean variables $\overline{y} = (y_1,...,y_m)$ for encoding the resulting action of the entire system.

Encoding States (Configurations) of the Overall System. Recall that a concurrent system consists of multiple sequential processes running in parallel. A state of the entire system, also called a *configuration*, is identified by the current states of its sequential components. Formally, the set of states of the system is a c-ary relation $S \subseteq S^1 \times ... \times S^c$, the initial state being $s_0 = (s_0^1,...,s_0^c)$. Therefore, S can be represented symbolically using the Boolean variables from $\overline{x^1},...,\overline{x^c}$. If $s = (s^1,...,s^c) \in S$, then $\lceil s \rceil(\overline{x^1},...,\overline{x^c}) = \bigwedge_{i=1}^{c}(\lceil s^i \rceil(\overline{x^i}))$. For clarity, we denote the set of states of the overall system by *Configurations*.

Supercombinators and Formats. As we mentioned in Section 2.1.3, supercombinators are rules for combining together actions of the individual sequential processes into event-outcomes of the overall system [Ros98]. Within a supercombinator, each process can participate with a visible event, a silent action τ, or not be involved at all. We denote the non-involvement with the symbol ε. For any alphabet Σ, we let $\Sigma^{\varepsilon} = \Sigma \cup \{\varepsilon\}$. In addition, the set of supercombinators is partitioned into existing *formats*, i.e., different configurations of switched on and switched off processes among $P_1,\ldots,P_c$. We denote by SC the set of supercombinators and by *Formats* the set of formats of the concurrent system.

Formally, the set of supercombinators can be represented as a $(c+3)$-ary relation $SC \subseteq Formats \times \Sigma_1^{\tau,\checkmark,\varepsilon} \times ... \times \Sigma_c^{\tau,\checkmark,\varepsilon} \times \Sigma^{\tau,\checkmark} \times Formats$, or more generally $SC \subseteq Formats \times (\Sigma^{\tau,\checkmark,\varepsilon})^c \times \Sigma^{\tau,\checkmark} \times Formats$. $(f_{src}, a_1, ..., a_c, a, f_{dest}) \in SC$ iff from a certain configuration and a certain format

f_{src} of the overall system, P_1 performs a_1, ..., P_c performs a_c and the overall system performs a switching to a format f_{dest}.

The operational semantics of the concurrent system can be considered an implicit LTS, whose transitions can be switched on and off:

- set of states - $Formats \times Configurations$
- set of labels - SC
- transition relation - $T \subseteq (Formats \times Configurations) \times SC \times (Formats \times Configurations)$. If the system is in a given configuration and in a given format, the individual processes transition relations determine if the labels are switched on or off. Formally, $(f_i,(s_i^1,\ldots,s_i^c)) \overset{(f_i,a_1,\ldots,a_c,a,f_j)}{\rightarrow} (f_j,(s_j^1,\ldots,s_j^c))$ iff
$(f_i,a_1,\ldots,a_c,a,f_j) \in SC \wedge \forall_{k=1}^{c}((a_k \neq \varepsilon \Rightarrow (s_i^k,a_k,s_j^k) \in T^k) \wedge (a_k == \varepsilon \Rightarrow s_i^k = s_j^k))$.

Encoding Supercombinators. For a given supercombinator $sc = (f_{src},a_1,...,a_c,a,f_{dest}) \in SC$, let $Passive(sc) = \{i \in \{1,\cdots,c\} | a_i = \varepsilon$, i.e. P_i is not involved in $sc\}$. Let $\overline{u} = (u_1,...,u_c)$ be a vector of (supercombinator-independent) Boolean variables. We denote:

$$lit(u_i) = \begin{cases} u_i & \text{if } P_i \text{ is not involved} \\ \neg u_i & \text{if } P_i \text{ performs a visible event or a } \tau \end{cases}$$

Note that a process might be switched on in a format and still be passive in a certain supercombinator in this format. Hence, we cannot use the format to conclude which processes are passive in a supercombinator.

Let $\overline{f}$ and $\overline{f'}$ be two vectors of $\lceil log_2|Formats|\rceil$ variables for encoding the source and destination format of a rule. Let $sc = (f_{src},a_1,...,a_c,a,f_{dest}) \in SC$. Then, $\lceil sc \rceil(\overline{y^1},...,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'}) = \bigwedge_{i \notin Passive(sc)}(\lceil a_i \rceil(\overline{y^i}) \wedge \neg u_i) \wedge \bigwedge_{i \in Passive(sc)} u_i \wedge \lceil a \rceil(\overline{y}) \wedge \lceil f_{src} \rceil(\overline{f}) \wedge \lceil f_{dest} \rceil(\overline{f'})$.

Hence, in an encoding of a supercombinator, we indicate a passive process P_i just by affirming a single Boolean variable u_i. We call u_i a *trigger*. For non-passive processes, we also encode the event that the process performs. The encoding of all supercombinators in all formats now becomes the following: $\lceil SC \rceil(\overline{y^1},...,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'}) = \bigvee_{sc \in SC} \lceil sc \rceil(\overline{y^1},...,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'})$.

Encoding a Transition of the Concurrent System. Let for $i \in \{1,\cdots,c\}$, $\psi_i(\overline{x^i},\overline{x^{i'}},\overline{y^i},u_i) :=$ if u_i then $(\overline{x^i} = \overline{x^{i'}})$ else $\lceil T^i \rceil(\overline{x^i},\overline{y^i},\overline{x^{i'}})$, where $\overline{x^i} = \overline{x^{i'}}$ is the short for $\bigwedge_{j=1}^{n_i}(x_j^i \Leftrightarrow x_j^{i'})$. The intuition behind a ψ_i is that, if P_i does not participate in a transition of the entire system, i.e. P_i is not involved in a supercombinator, P_i remains in the same state within its own labelled transition system OP^i. Otherwise, P_i progresses with respect to its transition relation T^i. Expressed as a Boolean formula, $\psi_i \equiv (u_i \wedge (\overline{x^i} = \overline{x^{i'}})) \vee (\neg u_i \wedge \lceil T^i \rceil(\overline{x^i},\overline{y^i},\overline{x^{i'}}))$.

We define a predicate $T^{Sys(c)}$ which is true exactly for the transitions of the overall system:

$$\begin{aligned} &\lceil T^{Sys(c)} \rceil(\overline{x^1},\cdots,\overline{x^c},\overline{x^{1'}},\cdots,\overline{x^{c'}},\overline{y^1},\cdots,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'}) = \\ &= \bigwedge_{i=1}^{c} \psi_i(\overline{x^i},\overline{x^{i'}},\overline{y^i},u_i) \wedge \lceil SC \rceil(\overline{y^1},\cdots,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'}) \end{aligned}$$

Encoding Fixed Length Executions of the Concurrent System. Within the BMC framework, let k be the maximal bound for the length of the counterexamples we are looking for. Then:

$\lceil Paths(Sys(c),k)\rceil($
// variables for P_1 $\quad \overline{x_0^1},\ldots,\overline{x_k^1},\overline{y_1^1},\ldots,\overline{y_k^1},u_1^1,\ldots,u_k^1$
// variables for P_2 $\quad \overline{x_0^2},\ldots,\overline{x_k^2},\overline{y_1^2},\ldots,\overline{y_k^2},u_1^2,\ldots,u_k^2$
$\ldots$ $\quad \ldots$
// variables for P_c $\quad \overline{x_0^c},\ldots,\overline{x_k^c},\overline{y_1^c},\ldots,\overline{y_k^c},u_1^c,\ldots,u_k^c$
// variables for the traces of the system $\quad \overline{y_1},\ldots,\overline{y_k},$
// variables for the formats in the rules $\quad \overline{f_0},\ldots,\overline{f_k})$

$=$ // processes start from their initial states and the initial format is Format[0]
$\bigwedge_{j=1}^{c}\lceil I^j\rceil(\overline{x_0^j}) \ \wedge \lceil I^f\rceil(\overline{f_0}) \ \wedge$
// supercombinators as transitions at each of the k steps
$\bigwedge_{i=1}^{k}\lceil SC\rceil(\overline{y_i^1},\ldots,\overline{y_i^c},\overline{y_i},u_i^1,\ldots,u_i^c,\overline{f_{i-1}},\overline{f_i}) \ \wedge$
// the idea of the ψ formulas - either transitions or wait, depending on the supercombinators
$\bigwedge_{\substack{j=1,\ldots,c\\ i=1,\ldots,k}}((u_i^j \wedge (\overline{x_{i-1}^j} = \overline{x_i^j})) \vee (\neg u_i^j \wedge \lceil T^j\rceil(\overline{x_{i-1}^j},\overline{y_i^j},\overline{x_i^j})))$

$=$
$\lceil I^{Sys(c)}\rceil(\overline{x_0^1},\ldots,\overline{x_0^c},\overline{f_0})\wedge$
$\bigwedge_{i=1}^{k}\lceil T^{Sys(c)}\rceil(\overline{x_{i-1}^1},\ldots,\overline{x_{i-1}^c},\overline{x_i^1},\ldots,\overline{x_i^c},\overline{y_i^1},\ldots,\overline{y_i^c},\overline{y_i},u_i^1,\ldots,u_i^c,\overline{f_{i-1}},\overline{f_i})$

5 Implementation Details

In the original version of BMC, the system is unwound step by step until the bound k is reached. Despite the recent advances in SAT-solvers' learning capabilities and incremental SAT-solving, we have observed that the bottleneck of the bounded refinement procedure is the SAT-solver. Therefore, we allow unfolding a configurable number i of steps of the process *Refinement* before running the SAT-solver. The SAT-solver is then used to check if *Refinement* can pass through the *sink* state in any of its last i unwindings. If yes, we have found a counterexample, otherwise we continue iterating until reaching the configured bound k. We refer to the value of i as *SAT-frequency*. We believe that this multi-step approach works well because the SAT-solver typically finds it much easier to find a satisfying assignment, if there is any, than to prove unsatisfiability, given CNF formulas with comparable size and structure.

We transform the Boolean formulae into equisatisfiable formulas in CNF using the Tseitin Encoding [BKWW08]. For brevity, we skip details about how we exploit the incremental SAT-interface. Currently, SymFDR supports MiniSAT (version 2.0), PicoSAT and ZChaff. For our test cases, we have found MiniSAT to be most efficient and all quoted results use MiniSAT. For our larger test cases, we also observed that MiniSAT finds a counterexample faster if we configure it to keep a smaller number of learned clauses and to restart more frequently. We also implemented adding unit learned clauses explicitly, as suggested in [ES03a]. Using positive polarity in decision heuristics also produced much better results.

The current implementation of SymFDR supports refinement checking systems with a single format only. However, we do not anticipate any problems generalising the problem to a multi-format setting. Moreover, most practical cases are also single-format.

In addition to the standard refinement check, SymFDR also supports the "Zig-Zag" temporal induction algorithm [ES03b], which makes BMC complete. However, due to concurrency, the

recurrence diameter is too big.

Some other approaches that did not scale well include exploiting unary rather than binary encoding, restricting the decision variables to the input ones [Sht00], incorporating PicoSAT's restarting scheme and phase saving strategy [Bie08] in MiniSAT, etc.

6 Experimental Results

In this section, we investigate the performance of SymFDR on a small number of case studies. We compare it to the performance of FDR, FDR used in a non-standard way, PAT [SLD08], and, in some cases, direct SAT encodings, NuSMV [CCG+02] and Alloy Analyzer [Jac06]. All SAT-based experiments use MiniSAT although SymFDR and the direct SAT encoder build upon MiniSAT version 2.0, while Alloy and NuSMV exploit the earlier version 1.14. All tests were performed on a 2.6 GHz PC with 2 MB RAM running Linux, except the test marked with a *, which was performed on a 4-MB-RAM PC running Linux, and the tests with PAT, which were performed on a 1.67 GHz PC with 2 MB RAM running Windows. The results are summarised in Table 1, Table 2 and Table 3. The last column titled $\sharp$ lists the length of counterexamples.

FDR-Div. The main search strategy for FDR is BFS [Ros94] because this has the combined advantages of always finding a shortest counterexample and of enabling implementations that work comparatively well on virtual memory. However, the strategy for discovering divergences is based on DFS. In test cases where it is likely that there are a good number of counterexamples, but that all of them occur comparatively deep in the BFS, there is good reason to use a bounded DFS (BDFS) algorithm to search for them, so that only error states reachable in less than some fixed number N of steps are reached. BDFS will quickly get to the depth where counterexamples are expected without needing to enumerate all of the levels where they are not. Provided that the counterexamples have something like a uniform distribution through the order in which the DFS discovers them, we can expect one to be found after searching through approximately $S/(C+1)$ states, where S is the total number of states and C is the number of counterexamples.

FDR does not implement such a strategy directly. It was, however, observed a number of years ago by Roscoe and James Heather that it is possible to use a trick that achieves the same effect using the present version of the tool. That is, arrange (perhaps using a watchdog) a system P' that performs only up to N events of the target implementation process P and then performs an infinite number of some indicator event when a trace specification is breached. Provided P is itself divergence-free, we then have that $P' \setminus \Sigma$ can diverge precisely when P violates the specification. FDR searches for this divergence by DFS.

This approach is particularly well suited to CSP codings of puzzles, since it is frequently known *ab initio* how long a counterexample will be, and the usual CSP coding uses the repeatable event *done* to indicate that the puzzle has been solved. The columns labelled FDR-Div in Table 1 and Table 2 report on the result of using this technique. In several ways this method is more similar to approach of PAT and SymFDR than the usual FDR approach. As is apparent from the experiments, there seems to be a large element of luck in how fast this approach is, possibly based on how close the path followed by the DFS is to a counterexample.

PAT. PAT [SLD08] is a model checker of a version of CSP enhanced with shared variables. Despite the BMC attempt [SLDS08], PAT is at present a fully explicit checker. In addition to LTL model checking, PAT supports CSP refinement checking which it performs in a way similar

to FDR although using DFS (instead of BFS), normalisation of the specification on-the-fly and partial-order reductions. In the test cases quoted here, the specification is given as a reachability property on the values of the shared variables. The reachability algorithm is based on DFS and state hashing is applied for compact state-space representation.

NuSMV. NuSMV [CCG+02] is a symbolic model checker verifying SMV against CTL properties using BDDs. The BMC framework of NuSMV, which we refer to as NuSMV-BMC, uses specifications written in LTL.

Alloy Analyzer. Alloy Analyzer [Jac06] is a fully-automatic tool for finding models of software systems designed in the lightweight Alloy modelling language. Alloy Analyzer could be considered a BMC checker due to its searching for a model only up to a certain scope and generating the model, if existing, using SAT-solving techniques.

Direct SAT Encodings. We believe that experimenting with direct SAT encodings of problems will offer guidance for optimising the translation of CSP to logic. For example, the chess knight test case suggests that a shorter chain of inference for high-level actions might be beneficial.

Test Cases. First, we consider the peg solitaire puzzle [Ros98], performing experiments on a chain of soluble boards with increasing level of difficulty. In the initial configuration, the board has all slots but one occupied by pegs. The only allowed move in the game is a peg hopping over another peg and landing on an empty slot. The hopped over peg is then removed from the board. The objective of the game is ending up with a board with a single peg positioned on the slot which had been initially empty. The length of any solution of the puzzle is equal exactly to the number N of pegs on the initial board — a hop event for $(N-1)$ pegs followed by an event *done* signifying a valid solution of the puzzle. The results are summarised in Table 1. The experiments indicate that for $N \geq 26$ SymFDR clearly outperforms FDR. In cases where a counterexample does not exist, FDR's BFS strategy outperforms the DFS-based tools PAT and SymFDR.

Our second test case is the chess knight tour. A knight is placed at position $(1,1)$ on an empty chess board of size $N \times N$. The objective is covering all squares of the board by visiting each square exactly once. Similarly to peg solitaire, a solution is generated as a counterexample to a specification asserting that the event *done* is never communicated. The length of a possible solution is N^2+1. The results are presented in Table 2. For $N=5$, FDR generates a counterexample faster, but for $N=6$ already, SymFDR found a solution in approximately 13 minutes, while FDR crashed after an hour and a half of state-space exploration.

The third test case — the classical puzzle of towers of Hanoi, aims primarily at comparing SymFDR with other SAT-based bounded checkers such as NuSMV and Alloy Analyzer. The results are summarised in Table 3. NuSMV-BMC and SymFDR seem to be competitive, both outperforming Alloy Analyzer. However, all non-SAT tools — FDR, PAT and NuSMV, are clearly of magnitudes more efficient than the SAT-based ones.

We can conclude that SymFDR is likely to outperform FDR in large combinatorial problems for which a solution exists, the length of the longest solution is relatively short (growing at most polynomially) and is predictable in advance. In those cases, we can fix the SAT-frequency close to a sizeable divisor of this length and thus spare large SAT overhead. The search space of those problems can be characterised as very wide (with respect to BFS), but relatively shallow — with counterexamples with length up to approximately 50–60. We suspect that problems with multiple solutions also induce good SAT performance. The experiments with the towers of Hanoi suggest that SAT-solving techniques offer advantages up to a certain threshold and weaken afterwards.

Table 1: Performance comparison – peg solitaire ($\sharp = N$)

N	FDR ♯ states checked	Time (sec.) FDR	FDR -Div	PAT	SymFDR SAT	SymFDR Total	SAT freq.	♯
20	41 703	0	0	6.14	6.92 10.82	10.23 14.06	10 20	20
23	411 976	5	0	2.16	11.21 6.62	16.72 12.33	12 23	23
26	4 048 216	72	0	7.23	27.73 15.39	35.16 24.66	13 26	26
29	28 249 254	581	1	89.93	54.29 39.33	65.39 49.61	15 29	29
32	$>$ 139 000 000 187 000 000*	$>$ 11 700 2 640*	5	8.91	175.05 172.56	189.20 186.61	16 32	32
35	—	—	1 485	399	529.88 291.59	548.80 309.94	18 35	35
38	—	—	43	1773.19	1 047.01	1 071.59	19	38
41	—	—	4	198.77	1 584.62	1 617.09	41	41

Table 2: Performance comparison – chess knight tour ($\sharp = N^2 + 1$)

N	FDR ♯ states checked	Time (sec.) FDR	FDR -Div	PAT	Direct SAT	SymFDR SAT	SymFDR Total	SAT freq.	♯
5	508 451	3	0.147	0.28	8.5	6.26 13.47	8.81 16.46	13 26	26
6	$>$ 120 000 000	$>$ 4 800	18	9.75	125.3	777.41	785.67	19	37
7	—	—	—	6.41	1 138	30 515.6	30 544.5	50	50

7 Conclusions and Future Work

In this paper we have demonstrated the feasibility of integrating a bounded refinement checker in FDR, and more specifically, exchanging the expensive explicit state-space traversal phase in FDR by a SAT check in SymFDR. On some test cases, such as complex combinatorial problems, SymFDR's performance is very encouraging, coping with problems that are beyond FDR's capabilities. In general, though, FDR usually outperforms SymFDR, particularly when a counterexample does not exist. We plan to further investigate and try to gain insight about the classes of problems that are tackled more successfully within the BMC framework.

We envision several directions for future work.

We plan to extend the BMC framework in SymFDR to make it applicable to the stable failures and failures-divergences models as well. This will involve extending the encoding of CSP processes with information about maximal refusals and divergences.

Table 3: Performance comparison – Hanoi towers ($\sharp = 2^N$)

N	Time (sec.)						SAT	$\sharp$
	FDR	PAT	NuSMV	SymFDR	Alloy	NuSMV-BMC	freq.	
5	0.198	0.64	0.43	4.92	11.53	2.157	16	32
6	0.202	2.89	0.66	27.26	327.37	34.910	32	64
7	0.164	5.01	0.171	182.68	21 537.27	1 864.75	64	128
8	0.182	27.76	0.292	3 114.05	—	2 218.23	128	256

We intend to implement McMillan's algorithm combining SAT and interpolation techniques to yield complete unbounded refinement checking [McM03]. This method has proven to be more efficient for positive BMC instances (instances with no counterexamples) than other SAT approaches. The completeness threshold in this case is the reverse depth of the state-space which is smaller than its recurrence diameter, as is the case with temporal induction [ES03b]. Moreover, experimental results have shown that, in practice, the algorithm often converges substantially faster, for bounds considerably smaller than the reverse depth. In addition, the interpolation algorithm allows jumping multiple time frames at once and hence allows tuning the SAT-frequency. The BMC framework presented in this paper will be the foundation to build upon.

Other avenues for further enhancing FDR's performance include partial-order reductions [Pel98] and CEGAR [COYC03, CCO+05].

Acknowledgements: We are grateful to D. Kroening and J. Worrell for their comments and P. Armstrong for his help with FDR. The analysis using DFS refinement through divergence checking was inspired by a correspondence several years ago between A. W. Roscoe and J. Heather. The work presented in this paper is supported by grants from EPSRC and US ONR.

Bibliography

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Pp. 193–207. Springer-Verlag, London, UK, 1999.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.* 98(2):142–170, 1992.

[Bie08] A. Biere. PicoSAT Essentials. *JSAT* 4(2-4):75–97, 2008.

[BKWW08] A. Biere, D. Kroening, G. Weissenbacher, C. Wintersteiger. *Digitaltechnik*. Springer, 2008.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic

Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. LNCS 2404. Springer, Copenhagen, Denmark, July 2002.

[CCO+05] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing* 17(4), 2005.

[CKOS05] E. Clarke, D. Kroening, J. Ouaknine, O. Strichman. Computational Challenges in Bounded Model Checking. *Software Tools for Technology Transfer (STTT)* 7(2):174–183, April 2005.

[COYC03] S. Chaki, J. Ouaknine, K. Yorav, E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proc. SoftMC 03*. 2003.

[EB05] N. Een, A. Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT05, volume 3569 of LNCS*. Pp. 61–75. Springer, 2005.

[ES03a] N. Een, N. Sorensson. An Extensible SAT-solver. In *SAT*. 2003.

[ES03b] N. Een, N. Sorensson. Temporal Induction by Incremental SAT-solving. In *Proceedings of First International Workshop on Bounded Model Checking*. ENTCS 4. 2003.

[G+05] M. Goldsmith et al. Failures-Divergence Refinement. FDR2 User Manual. Formal Systems (Europe) Ltd., June 2005.
http://www.fsel.com/documentation/fdr2/fdr2manual.pdf

[Hoa85] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21:666–677, 1985.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, CMU, 1993.

[McM03] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*. Pp. 1–13. 2003.

[Pel98] D. Peled. Ten Years of Partial Order Reduction. In *CAV '98: Proc. 10th International Conference on Computer Aided Verification*. Pp. 17–28. Springer-Verlag, London, UK, 1998.

[PY96] A. Parashkevov, J. Yantchev. ARC - a tool for efficient refinement and equivalence checking for CSP. In *IEEE 2nd International Conference on Algorithm and Architectures for Parallel Processing*. 1996.

[RGM+03] A. W. Roscoe, M. Goldsmith, N. Moffat, T. Whitworth, I. Zakiuddin. Watchdog transformations for property-oriented model checking. In *Proceedings of FME 2003*. 2003.
http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/91.pdf

[Ros94] A. W. Roscoe. *Model-checking CSP*. Chapter 21. Prentice-Hall, 1994.
http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/50.ps

[Ros98] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.
http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf

[Ros08] A. W. Roscoe. Lecture notes for the course *Advanced Concurrency Tools*. Oxford University Computing Laboratory 2008.

[RRS+01] A. W. Roscoe, P. Ryan, S. Schneider, M. Goldsmith, G. Lowe. *The Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.

[Sht00] O. Shtrichman. Tuning SAT Checkers for Bounded Model Checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*. Pp. 480–494. Springer-Verlag, London, UK, 2000.

[SLD08] J. Sun, Y. Liu, J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA*. Pp. 307–322. 2008.

[SLDS08] J. Sun, Y. Liu, J. S. Dong, J. Sun. Bounded Model Checking of Compositional Processes. In *Proceedings of the Second IEEE International Symposium on Theoretical Aspects of Software Engineering*. Pp. 23–30. IEEE, 2008.

Towards SMV Model Checking of SIGNAL (multi-clocked) Specifications

Julio C. Peralta and Thierry Gautier

INRIA, Centre Rennes-Bretagne Atlantique, 35042 Rennes cedex, France

Abstract: SIGNAL is a high-level data-flow specification language that equally allows multi-clocked descriptions as well as single-clocked ones. It has a formal semantics and is supported by several formal tools for simulation and static validation. This generality renders it useful for various specification, simulation, and verification tasks in embedded system design. SMV, in turn, is a language and model checker where synchronous models are single-clocked by definition. Roughly, we use standard techniques to describe clocks by Boolean variables, with the advantage that the number of such variables is kept to a minimum through a static analysis provided by the SIGNAL compiler. In particular, we propose a translation from possibly multi-clocked SIGNAL specifications into SMV specifications for their corresponding verification by model checking.

Keywords: Synchronous programs, Multiple-clocks, SMV, Model checking

1 Introduction

The increasing complexity of embedded systems and the costs associated with failures in their engineering and operation demand for models and tools that enable safe design and formal validation. In the past years, system design based on the *synchronous model* [BB91] has attracted the attention of many academic and industrial actors. This paradigm consists in abstracting the non-functional implementation details of a system, thus fostering a focused reasoning on the logic behind the instants at which the system functionalities should be secured. A benefit of designing with languages based on the synchronous model (e.g. ESTEREL [BG92], LUSTRE [HCRP91], or SIGNAL [LTL03]) is the availability of associated verification tools.

Among synchronous languages, a salient feature of SIGNAL is the notion of *polychrony*: the capability to describe systems in which components may have different clock rates. This expressivity coupled with its (compiler) ability to statically synthesize schedules (reasoning about the logic behind the source clock constraints) allows to embrace complex systems that arise in the form of GALS (globally-asynchronous locally-synchronous) or (loosely) time-triggered architectures, and thus renders the model checking of such specifications highly attractive.

SMV, in turn, is a language and model checker where synchronous models are single-clocked by definition. However, this apparent constraint does not prevent us from describing and verifying SIGNAL multi-clocked specifications as we demonstrate here. In order to reconcile both approaches (multi-clocked specifications and single-clocked ones), we use standard techniques to describe clocks by Boolean variables, with the advantage that the number of such variables is kept to a minimum through a static analysis provided by the SIGNAL compiler. Such analysis

produces a hierarchy of clocks (ordered by set inclusion) which is useful to avoid proliferation of SMV state variables.

The paper presents in Section 2 syntactic and semantic highlights of our source SIGNAL programs and the target SMV programs. Section 3, in turn, describes a generic SMV translation for each SIGNAL kernel operator. Then in Section 4 we provide examples of translations for (possibly multi-clocked) SIGNAL specifications and show the use of the SIGNAL compiler analysis to reduce the number of SMV state variables. The behaviours of the translated examples will be examined in Section 5 by model checking with SMV itself. Next, in Section 6 some elements for comparison with related work on model checking for other synchronous languages are presented. Finally, some concluding remarks and pointers for future work are given in Section 7.

2 SIGNAL and SMV: Syntax and semantics

In this section we introduce the SIGNAL kernel language and a subset of the SMV language used for our translation, as well as highlights of each language semantics.

2.1 SIGNAL kernel

SIGNAL is a data-flow relational language that relies on the polychronous model [LTL03, BGL08]. It handles possibly infinite sequences of typed values called *signals*. A signal `x` is implicitly indexed by discrete time, thus denoting the sequence x_t where $t \in H, H \subseteq \mathbb{N}$. At any instant (arbitrary $t \in \mathbb{N}$) a signal may be *present*, at which point it holds a value, or *absent*. There is no actual value associated with a signal when it is absent, by contrast with the instants when it is present. The instants of absence of a signal are denoted with the special symbol $\perp$, in the semantics. Signals may be of standard types, e.g. Boolean, integer, real, etc. Additionally, there is a particular type of signal called `event`. A signal of this type is always *true* when it is present. The set of instants (index set H above) where a signal `x` is present represents its *clock*, noted $\hat{x}$ (which is itself a signal of `event` type). A *process* is a system of equations (also called elementary processes) over signals that specifies relations between values and clocks of the signals. A *program* is a process. SIGNAL relies on a few primitive constructs that define *elementary processes* from which bigger processes may be built.

- *Function.* `y := f(x1,...,xn)` $\stackrel{def}{\equiv} x1_t \neq \perp \Leftrightarrow \ldots \Leftrightarrow xn_t \neq \perp \Leftrightarrow y_t = f(x1_t,\ldots,xn_t)$
- *Delay.* `y := x $ 1 init c` $\stackrel{def}{\equiv} x_t \neq \perp \Leftrightarrow y_t \neq \perp \Leftrightarrow [(t > 0 \wedge y_t = x_k \wedge k = \max\{t' \mid t' < t \wedge x_{t'} \neq \perp\}) \vee (t = 0 \wedge y_t = c)]$; (`c` is a compile time constant).
- *Undersampling.* `y := x when b` (where `b` is Boolean) $\stackrel{def}{\equiv} y_t = x_t$ if $b_t = true$, else $y_t = \perp$; (observe that expression `y := when b` is equivalent to `y := b when b`).
- *Deterministic merge.* `z := x default y` $\stackrel{def}{\equiv} z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.
- *Composition.* `P1|P2` $\stackrel{def}{\equiv}$ union of equations of `P1` and `P2`.

Table 1: Clock relations for primitives.

construct	clock relations
`y := f(x1,...,xn)`	$\hat{y} = \hat{x}_1 = ... = \hat{x}_n$
`y := x $1 init c`	$\hat{y} = \hat{x}$
`y := x when b`	$\hat{y} = \hat{x} \cap [b]$, $[b] \cup [\neg b] = \hat{b}$ and $[b] \cap [\neg b] = \emptyset$
`z := x default y`	$\hat{z} = \hat{x} \cup \hat{y}$

- *Hiding.* `P where x` $\stackrel{def}{=}$ `x` is local to the process `P`.

Derived operators are defined using the primitive operators above. For instance, a *synchronization* equation `x ^= y` specifies that `x` and `y` have the same clock. Moreover, the equation `x ^= y ^+ z` asserts that the clock of `x` is the union of the clocks of `y` and that of `z`. A *memory*: `y := x cell b init y0` allows to memorize in `y` the latest value carried by `x` when `x` is present or when `b` is *true*. Processes can be abstracted and declared, in a standard way, by explicitly designating their input and output signals (preceding their declarations with "`?`" and "`!`", respectively), with the sole constraint that the designated input signals cannot be defined inside such a process.

2.2 Static analysis of SIGNAL specifications

In order to assess the consistency of the clock relations associated with a program, and to organize the control of such a program, the compiler synthesizes a *clock hierarchy* [ABL95, BGL08]. A clock k_1 is said to be greater than a clock k_2 if k_2 is included in k_1 in terms of sets of instants.

Table 1 shows the *clock relations* associated with each primitive construct of SIGNAL. For the undersampling construct, the clock of the Boolean signal `b` is partitioned into $[b]$ and $[\neg b]$. The sub-clock $[b]$ (resp. $[\neg b]$) denotes the set of instants where the Boolean expression `b` is present and *true* (resp. *false*). Clock relations are automatically added and inferred by the compiler from any program to be analyzed. For a program `P = P1 | ... | Pn`, its associated clock relations are the result of applying the clock calculus on the conjunction of the clock relations associated with the sub-processes `P`$_k$, $k \in 1..n$.

The *clock calculus* [ABL94], in turn, seeks the greatest clock in the program, called *master clock*, from which all other clocks in the program can be extracted. In this case, the clock hierarchy is a tree. Nonetheless, in some programs, such a unique master clock may not exist. In this latter case, there are several local master clocks and the clock hierarchy is a forest. Note, however, that the root of a tree may not correspond with the clock of an input signal.

A program in which the clock hierarchy is a tree is *endochronous*. Such a program can be run in an autonomous way (its master clock plays the role of an activation clock). Otherwise, the program needs extra information from its environment to be run in a deterministic way.

The automatic code generation, for an endochronous program, relies on the synthesized clock hierarchy. Each clock is represented by a Boolean variable (booleanization stage [BBG+00]) which is true when the clock is present. For every signal, its value is meaningful when the

SMVpgr	→	*ModuleMain* | *ModuleStmt SMVpgr*	*AssignStmt*	→	ε | *lhs* `:=` *rhs* `;` *AssignStmt*
ModuleStmt	→	`MODULE id[(`*IdList*`)]`	*InvarStmt*	→	ε | *s_bool_exp*
		`VAR` *VarDclLst*	*lhs*	→	`init(id)` | `id` | `next(id)`
		`ASSIGN` *AssignStmt*	*rhs*	→	*cnst_exp* | *set_exp* | *case_exp*
		`INVAR` *InvarStmt*	*case_exp*	→	`case` *cases* `esac ;`
VarDclLst	→	ε | `id :` *Type* `;` *VarDclLst*	*cases*	→	`1 :` *rhs* `;` | *bool_exp* `:` *rhs* `;` *cases*

Figure 1: Subset of SMV language.

Boolean representing its clock has the value true. This allows to organize the control of the application following the clock hierarchy.

2.3 SMV: A subset

For our translation purposes we use only a subset of the SMV language. We present such a subset using the syntax of SMV that is compatible with the three versions [McM01, McM99, CCJ+05] of the language currently available on the web (SMV from CADENCE, SMV from CARNEGIE-MELLON UNIVERSITY, and NUSMV). We identify the following syntax with the oldest [McM01] of the three versions.

Syntax

Our SMV programs will consist of modules with parameters, except for the reserved module `main`. Module declarations may not be nested. Each module has a name, (possibly) a list of parameter names, and at most three sections: a section for variable declarations and/or module instantiation, marked at the beginning by the `VAR` reserved word; a section describing the variable values (initial, current or next instant), initiated by the `ASSIGN` keyword; and, a section describing invariants between the variables of the referred module, and whose beginning is marked by the reserved word `INVAR`. `DEFINE`, `FAIRNESS` and `SPEC` sections are not considered for the moment, but their use will be motivated when we present translation examples (Sect. 4), and some verification (Sect. 5) on them.

Figure 1 depicts the grammar of our subset of SMV. The possible type (*Type*) of an identifier (`id`) is `integer` (or intervals thereof), `boolean`, enumerated, or the name of another module; in this last case the identifier is used to refer to an instance of the referred module, and appropriate expressions should be given as parameters for the intended instance. Access to members of a module instance is through a dot notation (i.e. `id.var_id`).

An expression of an invariant is of type Boolean and may only contain module variables in its present form (i.e. no use of `init` or `next` operators are allowed). The right-hand-side of an assignment (*rhs*), for the case of a constant expression (*cnst_exp*), is a valid expression (e.g. containing arithmetic, Boolean or comparison operators) using any of the possible type values for a correct typing of the identifier in the left-hand-side; a right-hand-side, for the case of a set expression (*set_exp*), uses curly braces to extensively list the elements (separated by commas) of the desired set. Operations on sets are union and test of membership.

```
VAR
   f, h_x, h_y:  boolean;
ASSIGN
   init(y) := C;
   next(y) := case
                f & next(h_x) :  x;
                1 :  y;
              esac;
   init(f) := h_x;
   next(f) := f | next(h_x);
INVAR
   (h_y <-> h_x)
```

(a) `y := x$1 init C` in SMV

```
VAR
   h_x, h_y, h_b:  boolean;
ASSIGN
   init(y) := x;
   next(y) := case
                next(h_y) :  next(x);
                1 :  y;
              esac;
INVAR
   (h_y <-> (h_x & h_b & b))
```

(b) `y := x when b` in SMV

Figure 2: Function and undersampling operators in SMV

Semantics

Assignments for the first value (signaled by the use of `init` keyword on the *lhs*) of a program variable are only executed in the first instant of program execution, whereas assignments for the `next` instant are executed to obtain the value of the designated variable starting from the second instant. Assignments with not ocurrences of `init` or `next` in their *lhs* are executed at all instants. The order in which assignments are executed is given by the data dependencies existing between the variables occurring in the right-hand-sides of the assignments to execute (among all assignments of a program including those added by process instantiation). The rule that dictates the (partial) order of assignment execution says that a variable is first assigned before its value is used in a right-hand-side evaluation. Invariants define (possibly) extra relations/constraints to those already imposed by the assignments, thus limiting the valid executions of the source program to those where the invariant expressions hold.

3 From SIGNAL to SMV

Let us now describe a possible translation from simple equations in the SIGNAL kernel, to SMV module fragments. We will assume, for simplicity of exposition, that there is only on kernel operator per equation. Also, the translation for each such SIGNAL source equation is an SMV program fragment where variable declarations will be omitted (whenever possible) to allow for a greater translation generality, provided that their translation depends on whether they are input, output or local in the presence of multiple SIGNAL processes. Roughly, the translation has an SMV variable to carry the value of each source signal, as well as a Boolean SMV variable to denote its clock. The encoding of clocks as Booleans has already been used in other SIGNAL contexts [BBG+00], and more recently (outside SIGNAL), in abstractions of linear relationships (involving multiple clocks) for bounded model checking [CKY03, GG07].

Delay See SMV translation in Figure 2(a). Variables `h_x`, `h_y`, and `f` were added by the translation. The first two represent the clock of `x` and `y` respectively, while the last variable is

```
VAR
   h_x, h_y, h_z:  boolean;
ASSIGN
   init(z) := case
                h_x:  x;
                1:  y;
              esac;
   next(z) := case
                next(h_x):  next(x);
                next(h_y):  next(y);
                1:  z;
              esac;
INVAR
   (h_z <-> (h_x | h_y))
```

(a) `z := x default y` in SMV

```
VAR
   h_x, h_y, h_z:  boolean;
ASSIGN
   init(z) := f(x,y);
   next(z) := case
                next(h_z):  next(f(x,y));
                1:  z;
              esac;
INVAR
   (h_z <-> h_x) & (h_x <-> h_y)
```

(b) `z := f(x,y)` in SMV

Figure 3: Merge and function operators

used to detect the first instant of signal `x`. The guard labeled with `1` in the `case` statement is the default choice if none of the offered options holds. It is important to note here that the previous value of `x` should be kept (in its SMV definition, not shown here) in case it is absent (typically the default case in a `next` assignment) since this operator will refer to the previous value in SIGNAL semantics, which is not necessarily that of SMV. Also, note here that the value of `y` is kept in case its first instant does not coincide with that of SMV. Variable `f` is needed to detect the first instant of `y` (or `x` since they are synchronous).

For the SIGNAL kernel operators that follow we decided to keep the previous value of the defined variable, considering a general schema of translation, but in some particular occurrences of such operators we may not need to keep the value. The use of assignments that keep the value by default, allows for stuttering steps in our translation: a fundamental property if compositionality is desired.

Undersampling The SMV translation is depicted in Figure 2(b). Here we (potentially) need three clocks, one for each signal. The `init` definition fixes the value to that of `x` disregarding clock `h_y`. This is correct, however, because if `h_y` holds in the first instant then the value is correct, and if it doesn't then the value is not important, thus any value is valid in this last case.

Merge Figure 3(a) depicts the translation into SMV. Here, as above, we have three (clock) SMV variables. Once again, the initial assignment definition for the default case (labeled with `1`) appears arbitrary; it is justified, however, with a similar reasoning as that used for the undersampling operator above.

Function Refer to Figure 3(b) for the SMV translation. The reason for the initial instant assignment is similar to that used for the `when` operator above. Whether the output is present or not, the chosen value will be good.

3.1 Improving the translation into SMV

So far we have proposed an intuitively correct translation from SIGNAL elementary processes into SMV modules. We anticipate/conjecture that this translation is correct given the straightforward coding style of data-flow and clock constraints into SMV assign statements and invariants. Nonetheless, scalability is another desirable feature. To this aim we would like to reduce the number of SMV (state) variables introduced by our translation, since the number of such variables may (sometimes) render the state space exponentially bigger. The natural candidates for elimination are the clock variables, and perhaps also the SMV variables corresponding to signal source variables.

In order to avoid state variables in the translation the reader should know that SMV allows to define a variable as a function of other variables without the use of `next` or `init` operators. That is, such assignments may only refer to the present values of other SMV variables. In order to identify such variable definitions SMV provides a section named `DEFINE`. Roughly, uses of the variables so defined are replaced by their definition thus sparing some state variables.

At first sight, we may think that there is no need to introduce state variables for signals defined through operator `when`, or `default`, or `function`, since they all refer to values in the same instant. It would be tempting to replace them by their equivalent in the `DEFINE` section, and thus their values would be arbitrary when absent. However, this replacement would be incorrect when the values they define are referenced through a SIGNAL `delay` operator. Recall that the clock of a SIGNAL variable coincides with the instants of the associated SMV Boolean variable when it has value `true`, which is *not necessarily* the previous SMV instant. Consequently, for SIGNAL elementary processes using kernel operator `when`, `default`, or `function`, one may replace the SMV translation proposed above by one referring to present values in the corresponding `DEFINE` section.

For the SMV (clock) variables introduced we suggest to have one state SMV variable per tree root in the forest constructed (during clock calculus) by the SIGNAL compiler. The remaining SMV (clock) variables will be assigned in the `DEFINE` section. It is important to note here a shift in the translation. So far we translated clock relations as Boolean formulas in the `INVAR` section by pure constraint reasoning. Replacing such constraints with assignments (in the `DEFINE` section) renders the constraints *functional*. In summary, SMV variables that represent source SIGNAL clocks and are associated with an internal node in one of the trees found by the SIGNAL compiler may be translated using assignments in the corresponding SMV `DEFINE` section. In addition, the number of clock variables may be reduced by using one variable per synchronous equivalence class found by the compiler, as well as by elimination of those clocks (variables) found to be empty.

4 Translation examples

In the following we will provide examples of source SIGNAL specifications and their translation into SMV. Such SIGNAL examples will make part of a bigger specification describing a communication protocol for loosely time-triggered architectures [BCL+02].

4.1 A one-place FIFO

Consider a one-place FIFO in SIGNAL, Figure 4(a). The contents of `fifo_1` is the last value written into it. The output (signal `sx`) may only be read/retrieved after at least one instant that it was entered. The number of instants between a write and a read may increase non-deterministically. Each such instant is given by the (internal) clock of the local Boolean signal `b` (`interleave` process). Before translating into SMV we will give the `fifo_1` program to the SIGNAL compiler so that the hierarchy of clocks becomes evident as well as other optimisations applied by the compiler. For this program the compiler produces the SIGNAL program depicted in Figure 4(b). The hierarchy of clocks is made visually evident by the nesting of parallel[1] subprocesses (the only subprocess in this example comprises lines `5-13`). The root of the only tree is that of the clock defined at the top, line `3`. This line also indicates that the clock `h_b` is not fixed, but a free variable which could have any value at any instant. Line `4` gives the set of signals that share the same clock (`h_b`). Lines `6,8` indicate what is the name of the clock of signals `x, sx`, respectively, whereas lines `5,7` give their definitions. Finally, lines `9-11` provide the definition of the `fifo_1` output `sx` (through the use of the value of an intermediate variable `tmp`).

Now the translation of the compiled `fifo_1` program into SMV is in Figure 5. Translation of the negated delay spans lines `13-20`; translation of the `cell` operator lays between lines `5-12`; and, the `when` operator is translated into line `24` (if the type of `tmp` was not Boolean then a `case` statement would have been used). There is a clear depart from the translation schemes presented in Section 3. This stems from several improvements in the translation (already suggested at the end of Section 3), and with some conventions in the compiler program generation, Figure 4(b). A first convention exploited in our translation says that all uses of `when` operator have as first operand a synchronous expression (i.e. all its signals share the same clock) and second operand a signal denoting a clock *smaller or equal* to that of the first operand. As a result, our translation into SMV (Figure 2(b)) need not test the clock (`h_x`) of the first operand together with the clock (`h_b`) and value (`b`) of the second operand; it suffices to guard the use of the first operand value by the clock given as second operand (i.e. expression `h_y <-> h_x & h_b & b` becomes `h_y <-> h_b & b`). The next convention states that occurrences of the `default` operator have the standard form `x:= (a when h_f) default (b when h_g)` (with possibly more `default` and their corresponding operators) where `h_f, h_g` are clocks and `a,b` are synchronous expressions. Note here that clock signal `h_g` should be defined (implicitly or explicitly) as the difference between the clocks of `x` and `h_f`. Our translation of this operator (Figure 3(a)) won't have to translate the `when` operator occurrences in such equations, they serve to identify the clock guard for each `case` branch of the `default` operator. Finally, to discuss the `cell` operator recall that, in general, an equation `x := y cell z init C` is equivalent to the two equations `x := y default (x$1 init C) | x ^= y ^+ when z`. Uses of such operator have as second operand the clock of the defined signal. That is, `z` above will be a signal denoting the clock of `x`, and `y` is either a synchronous expression or a `when` operator.

An explanation of the simplification in the translation (lines `13-20`, Figure 5) of the negated delay (line `12`, Figure 4(b)) is in order. Because the definition of `bw` is quasi-circular (through a function operator and a one instant delay) we do not need the extra `f` variable to detect the first

[1] The SIGNAL parallel composition operator is commutative and associative.

```
process fifo_1 = (?  boolean x;
                  !  boolean sx;)
  (| sx := current_1(x,^sx)
   | interleave(x,sx) |)
  where
  process current_1 = (?  boolean wx;
                          event c;
                       !  boolean rx;)
    (| rx := (wx cell c init false)
                       when c
     |);
  process interleave = (?  boolean x,
                           sx; !)
  (| x ^= when b
   | sx ^= when (not b)
   | b := not(b$1 init false)
   |) where boolean b; end;
  end;
```

(a) One-place FIFO in SIGNAL

```
1:  process fifo1 = (?  boolean x;
2:                      !  boolean sx;)
3:   (| h_b := ^ h_b
4:    | h_b ^= tmp ^= b
5:    | (| h_x := when b
6:      | h_x ^= x
7:      | h_sx := when (not b)
8:      | h_sx ^= sx
9:      | sx := tmp when h_sx
10:     | tmp := (x when h_x) cell
11:             h_b init false
12:     | b := not (b$1 init false)
13:     |)
14:   |) where event h_b, h_sx, h_x;
15:         boolean tmp, b; end;
16: end;
```

(b) `fifo_1` after clock calculus

Figure 4: `fifo_1` source: Before and after applying clock calculus

instant, neither do we need an extra state variable (the `x` variable in Figure 2(a)) to guarantee that the delayed value is the correct one. All the information, clock-wise and data-wise, is comprised in the same signal, hence the compact SMV code generation. A straightforward generalisation of this reasoning allows us to translate in the same way all equations with form: `y := f(x$1 init C)`, where `f` is a SIGNAL function operator.

A two-place FIFO. Let us now consider the translation of the two-place FIFO resulting from composing two one-place FIFOs, as shown in Figure 6(a). For reasons of space we won't show the compiled version[2] of `fifo_2` but use the generated SMV code (Figure 5), and compose two

[2] The compiler automatically inlines all process instances.

```
1:  MODULE fifo_1(x,h_x)
2:  VAR
3:   h_b, b, tmp:  boolean;
4:  ASSIGN
5:   init(tmp) := case
6:                 h_x :  x;
7:                 1 :  0;
8:               esac;
9:   next(tmp) := case
10:                next(h_x) :  next(x);
11:                1 :  tmp;
12:              esac;
13:  init(b) := case
14:               h_b :  1;
15:               1 :  0;
16:             esac;
17:  next(b) := case
18:               next(h_b) :  !b;
19:               1 :  b;
20:             esac;
21: DEFINE
22:  h_x := h_b & b;
23:  h_sx := h_b & !b;
24:  sx := h_sx & tmp;
```

Figure 5: One-place FIFO in SMV.

```
process fifo_2 = (?  boolean x;
                  !  boolean xok;)
  (| xok := fifo_1(
                fifo_1(x))
   |) where
    process fifo_1 ...
     where
     process current_1 ...
     process interleave ...
     end;
   end;
```

(a) Two-place FIFO in SIGNAL.

```
MODULE fifo_2(x)
VAR
 ff11:  fifo_1(x);
 ff12:  fifo_1(ff11.sx);
INVAR
 ff11.h_sx <-> ff12.h_x
DEFINE
 xok := ff12.sx;
 h_xok := ff12.h_sx;
```

(b) Two-place FIFO in SMV.

Figure 6: SIGNAL and SMV: A two-place FIFO

instances of `fifo_1` accordingly. An interesting feature of the `fifo_2` SIGNAL program is that it is not endochronous (unlike `fifo_1`) and thus has multiple (master) clocks. Its translation into SMV (Figure 6(b)) uses the same schemas as for endochronous programs though. Yet another feature of the generated code is the existence of a clock constraint in the form of an SMV invariant. This expression was not translated as a clock definition since the SIGNAL compiler was unable to verify its validity, hence its form of constraint rather than a directed assignment (as those appearing in a `DEFINE` section, for instance).

4.2 The whole communication protocol

We've applied the mentioned simplifications for the complete specification of the protocol proposed by Benveniste et al. [BCL+02] (see ftp://ftp.irisa.fr/local/signal/publis/SIG2SMV/ for the whole protocol and its translation). Our simplification rules with the aid of the compiler reduced the number of state variables from 98 to 27 (disregarding any possible reductions in the SMV internal representation of such models), with the ensuing improvements in verification time.

5 Some model checking

Here we will pose some CTL [CGP00] queries (and LTL whenever possible, in order to ease the reading) to our previous SMV programs (Section 4) in order to elucidate some behaviour information from the SIGNAL source or the SMV translation. Also, our queries aim at illustrating the use of the SMV clock variables introduced by the translation.

5.1 The need for FAIRNESS constraints

Recall the `fifo_1` SMV module (Fig. 5). We are interested to know whether the SMV translation correctly assigns `true` for the first instant of signal `b`, given that the default case assigns `false` (i.e. `0`). Also recall that the first instant of an SMV program does not necessarily correspond to the first instant of some SIGNAL clocks. An LTL query could be as follows: `(!h_b U b)`. Our formula states that along *all* paths from the initial state(s) of the system our signal

may remain absent until it is first present with value 1. However, the SMV model checker says that our model fails to follow this LTL specification, and gives us a one-state trace to support such a response. A close examination of the counter-example shows that it is a state with a loop transition to it; that is, a behaviour of our system where the signal (h_b) is forever absent. This is a valid behaviour and is desirable for compositional reasons. For model checking, however, it is best to ignore behaviours consisting only of such self loops. Fortunately, SMV provides ways of ensuring that our queries are verified on behaviours excluding such infinitely stuttering behaviours. This may be achieved using SMV FAIRNESS statements to *restrict the verification to paths where such statements hold infinitely often*. Consequently, for our fifo_1 example, we added the following line: FAIRNESS h_b, and then our model verifies our LTL query above. Let us assume that appropriate FAIRNESS constraints have been added to all our examples and our LTL/CTL goals are to be verified along fair paths. Clearly the correct FAIRNESS statements refer to the clocks of the root(s) of the tree(s) found during clock calculus.

Now, we can check whether the Boolean (guard) signal (b) is alternating, by posing the LTL query: G(((h_b & b) -> X(!h_b U !b)) | ((h_b & !b) -> X(!h_b U b))). By such formula we mean that all states where the signal is present and true are always followed by a sequence of states where the signal may be absent until it first arises (is present) with value false, or the converse (for the signal values only). As expected, the SMV answer is affirmative.

5.2 Some non-determinism

Let us now query the fifo_2 module (Fig. 6(b)) where a stored value can only be retrieved (at least) two instants after it has been written, and not before. A CTL formula for inspecting whether given an input event (h_x), in the next instant, an output event (h_xok) is possible could be expressed as AG(ff11.h_x -> EX(h_xok)). For this goal the model checker answers *no* and gives a counter-example where every arrival of the output occurs two instants after an input was received. One may think that the output is *always* available exactly two instants after an input is placed, and thus pose the LTL query G(ff11.h_x -> X(X(h_xok))). Unfortunately this is not the case, as shown by another counter-example generated by SMV; the first output arrives four instants after the first input and then every three instants after another input. This (apparently) non-deterministic behaviour is due to the polychronous nature of the SIGNAL source by virtue of the two instances of the fifo_1 process (and more specifically, of the interleave process). Nonetheless we may assert that in general, there is always a behaviour for which after exactly two instants the output will arrive, in CTL: AG(ff11.h_x -> EX(EX(h_xok))). Alternatively, we may claim that given the input the output will always eventually arrive, in LTL: G(ff11.h_x -> F(h_xok))) and thus verify this with the model checker. Note that (in part) due to the imposed FAIRNESS constraints, given an input, the output will eventually arrive, even when the constraint is not on the input or output variables.

5.3 Correctness of the whole communication protocol

Before verifying the correctness of the protocol we succeeded in verifying the correctness of a claimed specification property (property number 16 [BCL+02]) of the protocol implementation:

never two writing events between two successive bus/buffer sampling events. Finally, we posed the same two CTL goals (to prove correctness of the protocol) to our SMV translation and thus confirmed the answer previously reported [BCL+02].

6 Related Work

Here we provide some comparison elements for work on model checking for three synchronous languages: ESTEREL, LUSTRE and SIGNAL, and work on model checking multi-clocked specifications outside the synchronous pardigm.

From the language expressivity perspective it is worth noting that ESTEREL and LUSTRE assume a master clock[3], while SIGNAL does not impose such a constraint. We may say that *the subset* of SIGNAL programs that are found to be endochronous by the compiler coincide with those synchronous programs with a single master clock.

For LUSTRE alone there is a model checker called LESAR [Ray06]. It is based on symbolic model checking too, and is able to reason about numerical constraints (convex polyhedra) on the transition systems, unlike SMV. However, LESAR is unable to validate liveness properties; only safety properties can be proved. A case study [BWL06] comparing model checking using LESAR and SMV (among other validation tools), shows the improved power of SMV compared with LESAR. In such comparison some translation from LUSTRE is used, but unfortunately it is not provided. Nonetheless, a close examination of the LUSTRE sources for their example shows that their programs were already single-clocked, and thus the translation into SMV appears much simpler than ours. Also, a manual translation from ESTEREL to LUSTRE is mentioned (not provided) to reach the facilities of SMV.

Two other transformations from LUSTRE to SMV are mentioned in [MMM05, MAWW05]. Neither of them provide the transformation rules used, nor the LUSTRE subset that could be translated.

For model checking ESTEREL programs we know of a proposal [MHM+95] that first translates such programs into an intermediate representation called *Boolean automata*, and then translates such programs into SMV. However, the actual definition and transformation into Boolean automata is not provided and it appears that not all such Boolean automata could be described in their version of SMV.

SIGALI [MRLS01] is the model checker for SIGNAL. It is tightly integrated with the (SIGNAL) compiler internal representation and optimisations. In addition to model checking, SIGALI is also useful for controller synthesis [BBG+01]. However, it does not generate counter-examples (nor witnesses). From the beginning, it was conceived as a decision procedure and some limited form of counter-example generation is possible for safety properties only. The problem of generating counter-examples may be cast as controller synthesis to somewhat project the source program on all the behaviours that lead to a given unsafe set of states. Such program projection may be interpreted as a set of counter-examples. As regards the input language, Sigali only supports Boolean and event signals whereas SMV has some (limited form of) integer reasoning and offers the possibility to bridge to bounded model checkers too. Nonetheless, in SIGALI, signal clocks need not always be explicit in LTL/CTL goals, they may remain implicit unlike our proposal for SMV.

[3] Esterel V7 appears to be multi-clocked, though.

We argue that making clocks explicit fosters a good understanding of the source specification, besides the potential feedback provided by counter-examples.

Outside the synchronous approach there is the work of Clarke et al. [CKY03] and the work of Ganai and Gupta [GG07]. In the context of bounded model checking, the former considers linear relations (equality and/or inequality) between clocks as input and then synthesises an automaton that describes all possible schedules of the clock ticks. Even though the system of linear relations may reference precise clock frequencies the synthesised automaton refers to logical instants, as in SIGNAL. Stuttering transitions are problematic for the automaton representation since it is not evident which is(are) the master clock(s), if any. The authors appear to circumvent the problem for their experiments but a definitive answer is missing. Their proposal is tightly dependent on their bounded model checker and the kind of properties checked appears to be restricted to safety issues only, whereas we are not dependent on safety properties and bounded model checking remains one possibility amongst several.

In a refinement of this work, Ganai and Gupta [GG07] propose a specialised translation of LTL goals for clocked specifications, which apparently render the bounded model checking scalable, for multiple-clocks. By contrast, we do not propose any model checking technique, neither an optimised translation of clocked LTL/CTL formulas. However, we propose a tightly integrated (with the SIGNAL compiler) translation from specifications with multiple clocks into SMV where bounded model checking is one option.

Last, but not least, SMV itself provides a syntax for composing (single-clocked) modules asynchronously, using the `process` keyword. This language feature offers the possibility to express some coarse-grained multi-clocked specifications without the need for extra explicit signaling (as is the case of our SMV Boolean variables to denote clocks). By contrast, in our source SIGNAL multi-clocked specifications clocks are finely interwoven. The challenge here is to derive a so-called GALS (globally-asynchronous locally-synchronous) description from the SIGNAL source multi-clocked specifications in order to match and profit from this SMV language feature (i.e. asynchronous composition of single-clocked modules).

7 Concluding Remarks

We have shown a simple source-to-source translation from SIGNAL (multi-clocked) specifications to single-clocked SMV programs for the purpose of CTL verification. Then we refine the translation taking into account the compiler analysis of the source SIGNAL program, in order to reduce the number of state variables added by the translation. This optimisation allows us to eliminate signal variables as well as (Boolean) clock variables. We stick to a syntax compatible with the three versions of SMV currently available. We presume soundness of our translation given the semantic proximity of the two languages and because the SMV coding neatly reflects the clock relations (using invariants or definitions) and data-flow (with assignments). The generality of our proposed translation is exercised through modeling and verification of SIGNAL specification with multiple master clocks. There are two additions to common/standard use of SMV and CTL for model validation, namely, *(a)* clocks are explicit in SMV and LTL/CTL goals, and *(b)* fairness constraints are needed for ensuring reactivity of the model behaviours; such constraints refer to the clocks of all the roots found during clock calculus.

Boolean SMV variables are used to model SIGNAL clocks. The translation automatically adds the Boolean clocks, and it is the user who will be responsible for a correct combination of clocks and signals while querying (in LTL or CTL) the produced SMV model. The chief condition for a sound use of explicit clocks is that *the value of a signal is only meaningful when its clock evaluates to* `true`. As a result, the user is only concerned with knowing the name of the clock of a signal and for every occurrence of the signal name (in a temporal formula) add the conjunct to test its presence (by referring to a true occurrence of its clock variable).

Future work. Here we only used the LTL/CTL verification functionality of SMV. We plan to experiment with other functionalities (bounded model checking, bounds analysis, refinement checking, induction, and compositional verification). In order to reduce the load to the user on combining clocks and signal values while querying the SMV model, we envisage automating the addition of clocks to temporal formulas without them.

Bibliography

[ABL94] T. P. Amagbegnon, L. Besnard, P. Le Guernic. Arborescent canonical form of Boolean expressions. Technical report 2290, Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France, 1994.

[ABL95] T. P. Amagbegnon, L. Besnard, P. Le Guernic. Implementation of the dataflow synchronous language SIGNAL. In *Conference on Programming Language Design and Implementation, PLDI95*. ACM Press, 1995.

[BB91] A. Benveniste, G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*. Volume 79(9), pp. 1270–1282. September 1991.

[BBG+00] L. Besnard, P. Bournai, T. Gautier, N. Halbwachs, S. Nadjm-Tehrani, A. Ressouche. Design of a Multi-formalism Application and Distribution in a Data-flow Context: An Example. In Gergatsoulis and Rondogiannis (eds.), *Intensional Programming II*. Pp. 149–167. World Scientific, 2000.

[BBG+01] A. Benveniste, P. Bournai, T. Gautier, M. Le Borgne, P. Le Guernic, H. Marchand. The SIGNAL declarative synchronous language: controller synthesis & systems/architecture design. In *Conference on Decision and Control*. Pp. 3284–3289. 2001.

[BCL+02] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, S. Tripakis. A Protocol for Loosely Time-Triggered Architectures. In *EMSOFT 2002*. Pp. 252–265. 2002.

[BG92] G. Berry, G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19(2):87–152, 1992.

[BGL08] L. Besnard, T. Gautier, P. Le Guernic. SIGNAL V4-INRIA version: Reference Manual. March 2008. http://www.irisa.fr/espresso/Polychrony/.

[BWL06] F. Boniol, V. Wiels, E. Ledinot. Experiences in using model checking to verify real time properties of a landing gear control system. In *Conference on Embedded Real-Time Systems*. January 2006.

[CCJ+05] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev. NuSMV 2.4 User Manual. ITC-irst, Via Sommarive 18, 38055 Povo (Trento), Italy, 2005. http://nusmv.irst.itc.it.

[CGP00] E. M. Clarke (Jr.), O. Grumberg, D. A. Peled. *Model Checking*. The MIT Press, 2000.

[CKY03] E. M. Clarke, D. Kroening, K. Yorav. Specifying and Verifying Systems with Multiple Clocks. In *International Conference on Computer Design (ICCD'03)*. P. 48. 2003.

[GG07] M. K. Ganai, A. Gupta. Efficient BMC for Multi-Clock Systems with Clocked Specifications. In *Asia and South Pacific Design Automation Conference*. Pp. 310–315. 2007.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*. Volume 79(9), pp. 1321–1336. September 1991.

[LTL03] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann. Polychrony for System Design. *Journal of Circuits, Systems, and Computers*, March 2003.

[MAWW05] S. P. Miller, E. A. Anderson, L. G. Wagner, M. W. Whalen. Formal Verification of Flight Critical Software. In *AIAA Guidance, Navigation and Control Conference and Exhibit*. August 2005.

[McM99] K. L. McMillan. The SMV language. March 1999. http://www.kenmcmil.com/smv.html

[McM01] K. L. McMillan. The SMV system (version 2.5.4). 2001. http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps

[MHM+95] M. Müllerburg, L. Holenderski, O. Maffeïs, A. Meceron, M. Morley. Systematic Testing and Formal Verification to Validate Reactive Programs. *Software Quality Journal* 4(4), 1995.

[MMM05] M. Moy, F. Maraninchi, L. Maillet-Contoz. LusSy: An open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems* 10(2-3):73–104, 2005.

[MRLS01] H. Marchand, E. Rutten, M. Le Borgne, M. Samaan. Formal verification of programs specified with SIGNAL: application to a power transformer station controller. *Science of Computer Programming* 41(1):85–104, 2001.

[Ray06] P. Raymond. Vérification de programmes synchrones avec LUSTRE/LESAR. In Navet (ed.), *Systèmes temps réel 1*. Pp. 181–216. Hermes science publications, Lavoisier, 2006.

Verification of safety requirements for program code using data abstraction

F.P.M. Stappers and M.A. Reniers

Department of Mathematics and Computer Science, TU/e,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract: Large systems in modern development consist of many concurrent processes. To prove safety properties formal modelling techniques are needed. When source code is the only available documentation for deriving the system's behaviour, it is a difficult task to create a suitable model. Implementations of a system usually describe behaviour in too much detail for a formal verification. Therefore automated methods are needed that directly abstract from the implementation, but maintain enough information for a formal system analysis.

This paper describes and illustrates a method by which systems with a high degree of parallelism can be verified. The method consists of creating an over-approximation of the behaviour by abstracting from the values of program variables. The derived model, consisting of interface calls between processes, is checked for various safety properties with the mCRL2 toolset.

Keywords: verification, safety requirements, translation, data abstraction, case study

1 Introduction

Subcontracting, buying off-the-shelf-components, and outsourcing are common in companies that develop and build embedded systems [Dja05, MAG+99, LNB+07]. These companies require high quality and fault free components. Regrettably, when integrating components from different suppliers, unforeseen errors occur or unexpected behaviour is encountered [RCK+92].

Since the number of components in industrial systems grow, more lines of code are needed to control the system's behaviour [SCK04]. To ensure that shipped systems are fault-free, tests are performed. Unfortunately, the absence of errors cannot be guaranteed by executing tests.

To prove the correctness of a system formal methods like model checking are needed [CGP00]. Usually, these models are built from the available documentation. However, if a system is developed under pressure (e.g. prototyping, limited resources, etc.) or hardly any information is available, the implementation often becomes the main source for deriving behavioural models.

Deriving models from documentation is hard. Creating usable models from source code is even harder. Without any abstraction techniques, the models are too big to be used for the analysis of behavioural properties. We observe that many relevant properties can be stated in terms of the interface calls between processes. By abstracting from internal actions, it is possible to combat state space explosions. Nevertheless, resulting models are still too large, because conditions (depending on values of program variables) determine if interface calls take place.

In this approach we also abstract from variables and assignments and therefore systematically

explore every alternative for every condition. This way an over-approximation of the possible interactions between the components is created, thereby preserving a simulation relation [GG89]. The approach can be used for verifying safety requirements [Lam77] on the interface communication for multi-disciplinary systems in the sense that any safety requirement that holds for the over-approximation also holds for the real system.

The goal of this paper is to assess the feasibility of the method sketched above by means of a case study. Since modern languages consist of many features and hierarchical structures, this paper assumes that the source code for the control software of multi-disciplinary systems is written in a simplified concurrency programming language (SCPL, Section 3). SCPL incorporates both the core features for describing concurrent imperative programs and the constructs found in the studied application. Note that we do not incorporate object oriented design issues such as classes, inheritance and templates. We conjecture that most contemporary programming languages can be translated to mCRL2, though this needs further investigation. In order to prove the practical value of the method it has been executed by hand on a large case study consisting of 236 parallel threads. Based on our case study, we see no problems in automation of the method.

Programs written in SCPL are transformed to models in mCRL2 [GMR+09] for which safety requirements are verified. To demonstrate the feasibility, the method is demonstrated on the implementation of a controller for a printer that manufactures Printed Circuit Boards (PCBs).

This paper is structured as follows. Section 2 gives a brief introduction to the relevant fragments of the language mCRL2 and the modal μ-calculus. Section 3 describes the translation from SCPL to mCRL2 used to acquire the model for the different components. In Section 4 and Section 5 the abstraction technique is applied on an industrial system. It describes the system and the framework on which the case study is demonstrated. The case study also demonstrates that we were able to prove useful requirements for this complex system. Section 6 discusses related work. Section 7 concludes with our results, discussion and future work.

2 Preliminaries

2.1 Syntax and semantics of mCRL2

An mCRL2 process is built from data-parameterised multi-actions and a collection of process operators. In this paper, a fragment of the syntax of the un-timed mCRL2 language is used. It is given by the following *BNF*:

$$\begin{array}{lcl} P & ::= & \alpha \mid P+P \mid P\cdot P \mid P \,\|\, P \mid \partial_B(P) \mid \Gamma_V(P) \mid X \\ \alpha & ::= & \tau \mid \mathsf{a}(\vec{d}) \mid \alpha \,|\, \alpha \end{array}$$

The small $\mid$ indicates a choice between symbols in the expression of the BNF. In this syntax α denotes a multi-action. A multi-action is a combination of actions combined by the big $|$. The empty multi-action is denoted by τ. An action consists of an action name and possibility a data parameter vector $\vec{d}$ (which is left unspecified). In this syntax processes are denoted by P. For processes $+$ denotes non-deterministic choice, $\cdot$ denotes sequential composition, and $\|$ denotes parallel composition. The operator ∂_B blocks all actions from set B of action names. Γ_V applies the communications described by the set V to a process. A communication in the set V

is of the form $\mathsf{a}_1 \mid \cdots \mid \mathsf{a}_n \rightarrow \mathsf{a}$. Application of Γ_V to a process means that any occurrence of the multi-action $\mathsf{a}_1(\vec{d}) \mid \cdots \mid \mathsf{a}_n(\vec{d})$ is to be replaced by $\mathsf{a}(\vec{d})$, for any $\vec{d}$. X is a reference to a process definition of the form $X = P$.

The semantics associated with an mCRL2 process, as used in the mCRL2 toolset, is a transition system where the transitions are labelled by multi-actions. A more elaborate description of the syntax and (timed) semantics are given in [GMR+07, GMR+09].

2.2 Modal μ-calculus

Modal μ-calculus formulae are used to describe behavioural properties. These properties can then be verified automatically against a behavioural model described in mCRL2. Modal formulae are specified in a variant of the modal μ-calculus extended with regular expressions [GM99] and data. The restricted fragment of the modal μ-calculus used in this paper is as follows:

$$\begin{array}{lll} \phi ::= & \mathit{true} \mid \mathit{false} \mid [\rho]\phi \mid \langle\rho\rangle\phi \\ \rho ::= & a \mid \rho \cdot \rho \mid \rho^* \\ a ::= & \mathsf{a}(\vec{d}) \mid \neg\mathsf{a}(\vec{d}) \mid \mathit{true} \end{array}$$

In this syntax, ϕ represents a property, ρ represents a sequence of actions and a represents the presence of a data parameterised action $\mathsf{a}(\vec{d})$, the absence of a data parameterised action $\neg\mathsf{a}(\vec{d})$, or any given action (represented as *true*) in such a sequence. The property *true* holds for any model, while *false* holds for no model. The property $[\rho]\phi$ states the property that ϕ holds in all states that can be reached by a sequence described by ρ. The property $\langle\rho\rangle\phi$ states the property that ϕ holds in some state that can be reached by a sequences described by ρ. To describe action sequences concatenation and iteration can be used. A more elaborate description of the μ-calculus and its semantics can be found in [Bra92, GM99].

3 Modelling the systems behaviour

Creating a model that preserves the essentials of a system, which is still useful for simulation or verification purposes is difficult. Depending on the requirements that are to be verified, different approaches and abstraction techniques need to be used. For large systems, the abstraction needs to be chosen such, that these properties can still be verified. In this paper, we try to verify safety requirements for a system, for which the behaviour is specified in more than 200 concurrent processes. If all statements are translated without a proper abstraction, it is merely impossible to verify properties due to the well-known state space explosion problem. Therefore a systematic method is required that transforms code into an useful model, appropriate for current model checking techniques [CGP00]. Because the requirements can be formulated in terms of interface calls between concurrent processes, the abstraction is performed on the internal operations of the individual processes. By abstracting from internal data (e.g. values of variables), conditions cannot be evaluated accurately. Therefore, conditionals are replaced by non-deterministic choices between the alternatives. This creates an over-approximation of the systems behaviour, because potentially more behaviour can happen. If a safety property holds for the over-approximation, it must hold for the real system. On the other hand, if a safety property does not hold for the over-approximation, it may still hold for the real system.

As indicated in the Introduction, we describe our approach in *Simplified Concurrency Programming Language* (*SCPL*). With SCPL it is possible to specify a parallel program, because it has a notion of concurrency. The syntax of SCPL is described by the following BNF:

$$
\begin{array}{lll}
\langle \texttt{program} \rangle & ::= & \langle \texttt{program} \rangle \, \langle \texttt{process} \rangle \quad | \quad \langle \texttt{process} \rangle \\
\langle \texttt{process} \rangle & ::= & \textbf{proc}\ C = \langle \texttt{statement} \rangle\ \textbf{return} \\
\langle \texttt{statement} \rangle & ::= & \textbf{call}\ N \quad | \quad \mathbf{x} := \mathbf{e} \quad | \quad \langle \texttt{statement} \rangle ; \langle \texttt{statement} \rangle \quad | \\
 & & \textbf{if}\ b\ \textbf{then}\ \langle \texttt{statement} \rangle\ \textbf{else}\ \langle \texttt{statement} \rangle\ \textbf{fi} \quad | \\
 & & \textbf{while}\ b\ \textbf{do}\ \langle \texttt{statement} \rangle\ \textbf{od} \quad | \quad \textbf{do}\ \langle \texttt{statement} \rangle\ \textbf{od} \quad | \\
 & & \textbf{suspend} \quad | \quad \textbf{resume}\ N
\end{array}
$$

A program consists of at least one process. A process consists of a unique identifier, the process identifier, and a body: a process with process identifier C and body of statements S is specified by means of **proc** $C = S$ **return**. It is assumed that each program contains a process with process identifier *init* that represents the process that is to be activated initially. The body of a process consists of statements that denote calls to other processes **call** N (where N is a set of process identifiers), multi-assignments $\mathbf{x} := \mathbf{e}$, sequential compositions $S; S'$, conditionals **if** b **then** S **else** S' **fi**, the (in)finite repetitions **while** b **do** S **od** and **do** S **od**; and statements **suspend** and **resume** N for the suspension and the continuation of (sets of) processes.

Let P_D denote the set all process identifiers, in which the identifier *init* specifies the initial process. Note that N (from **call** N and **resume** N) denotes a non-empty subset of P_D. This means that interface calls can only address processes that are known within the system.

We do not present a formal semantics of this language. In the upcoming sections, the programming constructs are given an informal semantics.

3.1 Translation Scheme

The translation function $\mathscr{A}$ takes a program written in SCPL and produces an mCRL2 specification, i.e., a tuple consisting of an initial process in mCRL2 and a set of mCRL2 process equations. For each process with identifier C in the SCPL program, there exists an equation defining a recursion variable X_C in mCRL2.

In the translation the following actions are used

- $Start_s(C)$ denotes a request for starting process C;
- $Start_r(C)$ denotes acceptance of the request for starting process C (by process C);
- $Done_s(C)$ denotes the return of process C (by C);
- $Done_r(C)$ denotes notification of termination for a run of process C;
- $Suspend_s(C)$ denotes the suspension of process C. If a process gets suspended the calling process interprets the suspend signal as the relevant part of the process is finished and the calling process can continue;
- $Resume_r(C)$ denotes the acceptance of the request to resume process C;

- $Resume_s(C)$ denotes the request to resume process C that is suspended.

Start, *Done*, *Suspend* and *Resume* denote the synchronizing actions between corresponding requests, which will be explained later in this section.

Assuming that the name of the initial procedure is *init*, the translation function $\mathscr{A}$ is defined as:

$$\mathscr{A}(p_1 \cdots p_k) = (\partial_{Bl}(\Gamma_E(\Gamma_B(Start_s(init) \cdot Done_r(init) \parallel (\Big\|_{C \in P_D} X_C)))), \bigcup_{i=1}^{k} \mathscr{A}'_{\chi_i}(p_i))$$

where

- P_D contains the identifiers of all processes defined in the program.
- $Bl = \{Start_s, Start_r, Done_s, Done_r, Resume_s, Resume_r, Suspend_s\}$ denotes the set of blocked actions.
- $B = \{Start_s \mid Start_r \rightarrow Start,\ Done_s \mid Done_r \rightarrow Done\}$ denotes primitive communications.
- $E = \{Suspend_s \mid Done_r \rightarrow Suspend,\ Resume_s \mid Resume_r \rightarrow Resume\}$ denotes additional communications.
- $\Big\|_{j \in J} X_j$ describes the processes running in parallel and is recursively defined as:

$$\Big\|_{j \in \emptyset} X_j = \tau, \qquad \Big\|_{j \in J \cup \{k\}} X_j = X_k \parallel \Big(\Big\|_{j \in J \setminus \{k\}} X_j\Big).$$

- the sets χ_i of process identifiers are pairwise disjoint and disjoint from the set of recursion variables used to capture the processes that are defined in the program.
- $\mathscr{A}'$ specifies the translation function for processes which is defined in the rest of this section.

The encapsulation operator ∂_{Bl} and communication operators Γ_E and Γ_B are applied to the parallel composition of all processes to synchronize successful interface calls between processes and to block individual non-successful interface calls. The different local communication operators Γ_E and Γ_B are required to guarantee unique solutions. For example, $\Gamma_{\{a|b \rightarrow c, a|d \rightarrow e\}}(a|b|d)$ has multiple outcomes, namely $c|d$ and $e|b$.

Each process of the program is associated with at least one mCRL2 process equation by means of the translation function $\mathscr{A}'_{\chi_i}$: one of these corresponds to the translated process, while the others are introduced to capture repetitions in the body of a process. To ensure that all introduced recursion variables differ from other recursion variables, the translation function is parameterized by a set of recursion variables χ_i that are free to be used and are chosen sufficiently large.

We assume that the initialization process *init* can only be called from outside the system.

3.2 Processes

Processes decompose the system's functionality into smaller manageable parts, where each process carries out a specific task. If a task is too complex for a process it is often refined by invoking other more basic processes. The behaviour of a process can be implemented as a function, subroutine, procedure or some functional behaviour. The behaviour of an individual process is defined by statements placed in some order.

Let **proc** $C\ = S$ **return** denote the implementation of a process, where C defines the process identifier and S defines the control flow and data transformations. A process can be invoked by using a call and when the process completes the set of tasks it will notify the calling process with a return.

In SCPL all processes that can be addressed are defined in the program. A process can be either busy (by performing tasks) or idle. A busy process can become temporarily idle, until another process addresses the suspended process to continue. For processes that not have been suspended (e.g. are idle), the resume will not activate the execution of a process (e.g. the process stays idle). Let $\mathscr{A}'_\chi$ denote the translation function for a process, where χ denotes the set of available recursion variables, then the translation function for process identifier C and statement s is given by:

$$\mathscr{A}'_\chi(\textbf{proc}\ C\ = S\ \textbf{return}) = \left\{ \begin{array}{rl} X_C = & Start_r(C) \cdot t_p \cdot Done_s(C) \cdot X_C \\ + & Resume_r(C) \cdot Done_s(C) \cdot X_C \end{array} \right\} \cup E_p$$

where $(t_p, E_p) = \mathscr{A}''_{\chi,C}(s)$ and $\mathscr{A}''_{\chi,C}$ is the translation function for statements as defined in the following subsection. The first summand of the equation specifies the starting of the process. The second summand is used to reflect the call for resuming the process. The translation function is parameterised by the identifier C of the process that is being translated. This identifier is later used to notify that a process is suspended.

3.3 Statements

In this subsection the transformation $\mathscr{A}_{\chi,C}$ of statements is discussed. Let p and q denote statements and b a Boolean expression.

Interface calls An interface call contains a non-empty set of process identifiers. If the set contains one element, it behaves as a call to a single process. If the set contains more elements, it behaves like a call to multiple process, which need to be executed concurrently. A call simultaneously enables the start of the processes referred to in the set N. Processes can only be started if they are idle. If a call is addressed to a busy process, the call is postponed until the process becomes idle after completing the current task entirely. Thus, for processes that are temporarily idle the call is also postponed. A process, that performs a call, resumes after all called processes have either suspended or completed their tasks. The interface call statement is translated as follows:

$$\mathscr{A}''_{\chi,C}(\textbf{call}\ N) = \left(\Big|_{n \in N} Start_s(n) \cdot \Big|_{n \in N} Done_r(n), \varnothing \right)$$

where $\Big|_{n\in N}\ p(n)$ is inductively defined as:

$$\Big|_{n\in\varnothing}\ p(n) = \tau, \qquad \Big|_{n\in N\cup\{k\}}\ p(n) = p(k) \mid \Big|_{n\in N\setminus\{k\}}\ p(n).$$

Since there is no need to introduce additional process equations, the second element of the tuple is empty.

Assignments The multi-assignment statement $\mathbf{x} := \mathbf{e}$ defines the atomic value update for the variables $x_1,\ldots,x_n$ with the values of $e_1,\ldots,e_n$. As discussed earlier, we choose to abstract from variables and the assignments to them. When translating a multi-assignment it is translated as follows:

$$\mathscr{A}''_{\chi,C}(\mathbf{x} := \mathbf{e}) = (\tau, \varnothing)$$

where the assignment itself is translated to an internal non-observable action and there is no need for additional process equations.

Sequential composition Almost every imperative programming language allows the execution of statements in a sequential order. It is evident, that the control flow depends on the sequential order and needs to be preserved. The translation for the sequential composition is as follows:

$$\mathscr{A}''_{\chi,C}(p\ ;\ q) = (\mathscr{A}''_{\phi,C}(p)\cdot\mathscr{A}''_{\psi,C}(q), E_p\cup E_q)$$

where ϕ and ψ are sets of recursion variables such that $\phi\cap\psi=\emptyset$ and $\phi\cup\psi\subseteq\chi$. These sets can always be chosen large enough to allow for the subsequent translations to have enough fresh recursion variables available.

Conditionals The evaluation of a conditional depends on the values of a set of variables. By abstracting from the values of variables, it is impossible to determine the outcome of a condition. Therefore, conditionals are modelled as non-deterministic choices. The conditional statement is translated as follows:

$$\mathscr{A}''_{\chi,C}(\textbf{if } b \textbf{ then } p \textbf{ else } q \textbf{ fi}) = (\mathscr{A}''_{\phi,C}(p) + \mathscr{A}''_{\psi,C}(q), E_p\cup E_q)$$

where ϕ and ψ are sets of recursion variables such that $\phi\cap\psi=\emptyset$ and $\phi\cup\psi\subseteq\chi$.

Loops Loops are used to repeat statements that need to be carried out several times in succession. Loops are either used for computational purposes (for which they need to be finite) or for controlling the control flow (possibly infinite).

Both loops and infinite loops are modelled by means of recursion variables. If a control loop is finite, it has a condition which determines whether or not to abort the loop. Such a conditional choice is modelled as a non-deterministic choice (as is the case for conditionals). Of course, for infinite loops, there is no reason to introduce such non-determinism.

The reason for having infinite loops in SCPL is that virtually all systems have a part that needs to run continuously during executing and for which it is not possible to abort this process. In

these circumstances it must not be possible to end the control flow. An infinite loop and a finite loop are translated as follows:

$$\begin{aligned}\mathscr{A}''_{\chi,C}(\textbf{do } p \textbf{ od}) &= (Y, \{\, Y = t_p \cdot Y \} \cup E_p)\\ \mathscr{A}''_{\chi,C}(\textbf{while } b \textbf{ do } p \textbf{ od}) &= (Y, \{\, Y = t_p \cdot Y + \tau \} \cup E_p)\end{aligned}$$

where Y denotes a fresh recursion variable from χ, and t_p and E_p are defined as $(t_p, E_p) = \mathscr{A}''_{\chi\setminus\{Y\},C}(p)$. Note that an additional τ is required for a finite loop to make a non-deterministic choice between loop continuation or loop termination.

Processes suspension If a system runs multiple parallel processes, it is often desired to suspend the execution of a process before it may proceed. For this reason SCPL has a statement that suspends a process.

$$\mathscr{A}''_{\chi,C}(\textbf{suspend}) = (Suspend_s(C) \cdot Resume_r(C), \varnothing).$$

Process continuation Processes that are suspended become (temporarily) idle, until another process requires the continuation. SCPL offers a solution, that enables the continuation of a suspended processes by means of a resume statement. Note, that when a process is idle (after completing a task), it cannot be resumed or started by a resume. As for interface calls, multiple processes can be resumed (in parallel) by a single resume. The resume statement is translated as follows:

$$\mathscr{A}''_{\chi,C}(\textbf{resume } N) = \left(\Big|_{n\in N} Resume_s(n) \cdot \Big|_{n\in N} Done_r(n), \varnothing \right).$$

To illustrate the translation, consider the following example.

Example 1 (Translation by example) *Consider a system with two concurrent processes init and P. Process P has two computational parts, that should be suspended in between. init calls P and waits until P finishes the first computational part. When finished, the init process resumes P in order to execute the second part.*

proc *init*	=	**call** *P;*	**proc** *P*	=	**b**:= *true;*
		resume *P;*			***suspend;***
		return;			**b** := *false;*
					return;

After applying the transformation we obtain the following mCRL2 specification:

$$\begin{aligned} C_{init} &= Start_r(init) \cdot Start_s(P) \cdot Done_r(P) \cdot Resume_s(P) \cdot Done_r(P) \cdot Done_s(init) \cdot C_{init}\\ &+ Resume_r(init) \cdot Done_s(init) \cdot C_{init}\\ C_P &= Start_r(P) \cdot \tau \cdot Suspend_s(P) \cdot Resume_r(P) \cdot \tau \cdot Done_s(P) \cdot C_P\\ &+ Resume_r(P) \cdot Done_s(P) \cdot C_P \end{aligned}$$

with the following initialization:

$$\partial_{Bl}(\Gamma_E(\Gamma_B(Start_s(init) \cdot Done_r(init) \,\|\, C_{init} \,\|\, C_P)))$$

The corresponding labelled transtion system (where internal τ actions are hidden) is depicted as:

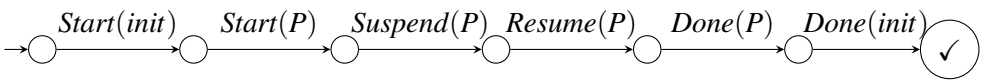


Figure 1: The labelled transition system for processes *Init* and *P*

4 Industrial application

To test the approach, it is applied on an industrial system, called "Lunaris" [Roo07]. The Lunaris is an Etch Resist Printer, intended to operate in the manufacturing of printed circuit boards (PCBs). In current PCB production processes, the substrate is laminated with a photo resist and using a lithographic process the desired photo mask is created on the substrate. With the development of the Lunaris, it is possible to skip the expensive task of creating the mask that is required for illuminating the photo resist. By directly printing the resist in the desired pattern, it is possible to create customized and individual PCBs at lower costs.

This prototype printer has been developed for one year and has been extensively tested within this period. While the system has many physical components, we limit ourselves to verify behavioural system requirements at the level of the controller. At controller level, the Lunaris consists of 245 multi-threaded tasks (running in parallel) that are implemented in C# [AW02]. The tasks specify behaviour for amongst others logging and error handling. In total 170.000 lines of code are needed to implement the behaviour. The code is distributed over 120 classes in 40 files.

Translating the code to mCRL2 directly is possible, however this will make any exhaustive verification technique useless. A brief analysis shows that more than 10^{1000} transitions are needed, if we want to incorporate all behaviour. For this reason we apply the abstraction techniques proposed in Section 3.

The Lunaris has 7 different axes over which mechanical components move. The areas in which they operate overlap each other. If they operate in the same area they collide and cause damage to the system. By means of special rules in the controller this should be prevented.

The controller must be defined by using a predefined set of tasks. In turn, these accessible tasks can execute other tasks, which are not directly available to the controller. Since we do not have access to the implementation of the controller, we allow the accessible tasks to be executed in arbitrary order, when performing the verification.

Different tasks run in parallel, but it is not possible to run the same task simultaneously multiple times. Every task belongs to a certain activity type, e.g. logging, error handling, time delaying, ignore errors, operate hardware, etc.

The tasks are called via a master-slave protocol. A task is a master if the task itself requests execution of another task. A task is a slave if another task requests its execution. We assume that the communication takes place over non-lossy channels. The following message types are communicated between tasks:

Start A Master wants to start a task on a slave.

Done A Slave indicates that a task has been successfully terminated.

Resume A Master wants to resume a task on a slave.

Suspend A Slave suspends the current process and notifies the master.

By means of several simplifications we obtain processes described in SCPL from the C# code for tasks. First, we only consider the tasks that are relevant for manufacturing a product. Therefore, nodes that are exclusively used for logging and error-handling local to components are excluded. Error-handling nodes are used to detect susperious implementation details. This can be decided based on the activity types of the tasks. Second, we only consider "good weather" behaviour. "Good weather" behaviour is the assumption that the components behave without faults. This means that a printhead is not broken, the system prints when it is supposed to, communication channels are not lossy, etc. Third, we assume that protocols used for communication are handled correctly by the framework and the embedded software is implemented according to the specification. For this reason we do not have to specify and verify the software that provides the communication or the software on the embedded systems. Fourth, the execution of a task requires a certain amount of time. We decide not to model time aspects. This way, we prove that the correctness of the controller is not affected by performance. Note that this decisions prohibits us to verify performance properties. Fifth, for the initialization we assume the system is turned off and all components are positioned such that they reside in their initial position.

After these simplifications, we obtain 236 single threaded process, running in parallel. Based on their type of behaviour, the task templates can be decomposed into "execute tasks" and "switch tasks".

4.1 Execute Tasks

An execute task is a task that is executed once. An example of an execute task is moving the printhead device to a given position. When started, an execute task automatically completes after a finite amount of processing time.

The semantic structure of an execute task becomes as the hierarchical state machine depicted in Figure 2. A sub-task is indicated by a rectangle. Single lined boxes indicate that the sub task consists of a single state. Double lined boxes indicate that the sub task is a hierarchical state machine, which can send and receive different types of calls.

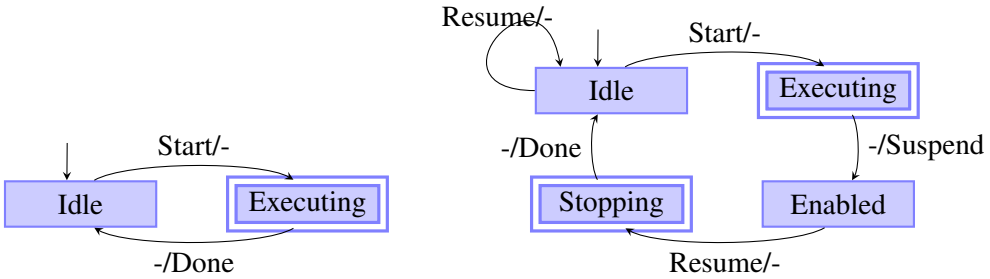


Figure 2: State diagrams of an Execute Task and Switch Task

The behaviour of an execute task with identifier C can be mapped to a process of the form:

proc C = "Executing" **return**

4.2 Switch Task

A switch task is a task that whenever started, needs to be stopped explicitly. Switch tasks are often used to enable hardware components (e.g. enabling controllers if the system has reached a certain run-level).

If a switch task is invoked, it first executes some behaviour, after which it comes into a stable enabled state. There it waits, until it receives an external signal to resume and finalize the task. Tasks that call a switch task, continue after the called switch task reaches the stable enabled state.

A switch task can be mapped to the hierarchical state machine depicted in the right of Figure 2. The behaviour of a switch task with identifier C is mapped to a process of the form:

proc C = "Executing"; **suspend**; "Stopping" **return**

5 Verification of framework properties

To validate that the technique can be used to verify safety requirements, this section discusses the requirements that have been verified with help of the obtained mCRL2 specification and the results. The formulae presented are in action terms of an mCRL2 specification.

To ensure correct behaviour, safety rules are formulated for the architecture. A safety rule is a rule for which the condition must hold, before the Lunaris can perform a specific action. The Lunaris has two kinds of safety rules. The first set consists of eight rules which represents warnings. If such a rule is violated, the system will raise a warning, but will continue to operate. These safety requirements are of the form: The "Switch Task (ST)" must be running if the "Execute Task (ET)" is executed. Such properties can be expressed by the following formulae template:

- An execute task ET may not be started before the switch task ST is "Enabled":

$$\begin{array}{ll} & [(\neg Suspend(ST))^* \cdot Start(ET)]false \\ \wedge & [true^* \cdot Start(ST) \cdot (\neg Suspend(ST))^* \cdot Start(ET)]false \end{array}$$

- An execute task ET may not be stopped after the switch task ST is being stopped:

$$\begin{array}{ll} & [true^* \cdot Start(ET) \cdot (\neg Done(ET))^* \cdot Resume(ST)]false \\ \wedge & [true^* \cdot Start(ET) \cdot (\neg Done(ET))^* \cdot Done(ST)]false \end{array}$$

The analysis shows that three of the eight safety rules are superfluous, i.e., the warning can not arise in any behaviour of the system (and hence do not occure if the controller implementation is without any flaws).

The second class of rules consists of 30 safety properties which only allow the execution of a task (T), if it is safe to do so. These tasks involve the movement of the printhead calibration system (PCS) or shuttle (Shuttle). Since they physically operate in each other's workspace, it is possible that the system can incur physical damage if such a safety property is violated. As

a result the system halts, if a rule is violated. These tasks may only be started if the printhead calibration system activate-switch task is idle or is started, its scan-task is idle and the tasks for moving the shuttle are idle. To verify that the rules are valid throughout execution, temporal logic formulae of the conjunction of the following forms have been constructed (S, T, and U are actions),

$$[true^* \cdot S \cdot (\neg T)^* \cdot U]false$$

where T is the task that may not be executed between tasks S and U.

All of the formulae have been checked and four requirements are violated in the model. Since the verification has been performed with a controller that is in no way restricted, it can be concluded that the safety rules are never violated, if the implementation is according specification.

In order to verify temporal formulae the specification is linearised to a linearised process specification. For all the requirements, this linearization step has been executed just once. This took approximately 53 minutes on a computer with a Intel® Pentium® D930 processor and 2 Gb RAM running Linux. The subsequent verification for a single requirement took less than 15 minutes.

6 Related work

To determine if a system is free from programming bugs, inconsistencies, run-time errors, or non-portable constructs various tools like LINT [Dar86, Joh78], POLYSPACE [Inc], and QA-C++ [Gro] can act as an extension to standard debuggers. When it comes to the verification of dynamic properties (deadlocks, unexpected behaviour) tools like Java PathFinder [ws08] or StEAM [LME04] can be used. These tools use a virtual machine in which models are translated to byte code, and executed afterwards to verify properties. Unfortunately, the size of the code is related to the underlying state space that needs to be explored, e.g., it becomes harder, or even impossible to verify safety properties. As stated by Java PathFinder: "While software model checking in theory sounds like a safe and robust verification method, reality shows that it does not scale well."

One can argue that the work presented here is comparable to the theory of abstract interpretation. In abstract interpretation [NNH99], abstract values are chosen for variables. Behavioural models obtained via this approach depend on the (initial) values of data variables. Consequently, it requires manipulation of the data variables. For relatively small systems, this method can be fruitful. However, for larger systems, this may lead to a state space explosion, due to the number of parallel processes combined with the number of possible abstract data values. In order to verify larger systems, either a more coarse grained abstraction is required (thereby losing information) as we do in this paper or state space reduction techniques (symmetry reduction, bisimulation reduction, etc.) need to be applied. Since almost every thread specifies unique behaviour, we could not benefit greatly from symmetry reduction. The application of bisimulation reduction techniques requires the generation of the underlying state space. Without any abstraction techniques, the approximated size of the state space for this system should roughly be 10^{1000} states.

Work related to our method can also be found in the Bandera toolset [CDH+00]. The Bandera toolset translates Java source code to a model, which is used to verify properties about the system by model checking techniques. Unfortunately, large (software) systems lead to state spaces that are beyond today's computational power. The Bandera toolset itself, only accepts closed code. For this reason the system needs to be complete before it can be verified. With help of extensions it is possible to verify open systems (e.g. an environment generator for Bandera [TDP03]), but it still requires a full and correct implementation of a source code unit. Since our method abstracts from variables we can deal with partly implemented units and code skeletons.

The author of [Kof07] presents a way for checking component behaviour compatibility, written in behaviour protocols and checked with the Spin model checker afterwards. Using LTL formulae, they manage to verify properties on a well documented system of 20 components. In our case study we tackled a bigger system running 230 concurrent processes, and performed a successful verification with a different toolset. Next to that the semantics of our components differs: we cope with processes that can be suspended and need to be resumed afterwards, while the components mentioned in [Kof07] do not facilitate this mechanism.

Work presented in [Hol01] shows a method for directly deriving a Promela specification from C code. This technique creates for every command a corresponding action in a Promela specification. In [Web07] another approach is taken with Promela. Here experiments are conducted with a virtual machine based approach for state space generation. By evaluating the byte-code language, they provide a way to efficiently execute operational semantics for modelling and programming languages. Undoubtedly, these techniques perform well on small toy examples for examining specific code constructs. However when changing the scope from specific code constructs to the control flow for examining larger concurrent systems, more rigorous techniques are required. In that sense, the method described in this paper can be viewed as an extension to their techniques.

Notice that our work shares resemblance with SLAM by Ball et al. [BR01]. One of the SLAM approaches is based on refining the abstraction which turns software implementations into boolean programs [BR00]. The basic idea is to leave out data initially, and include it when needed later on. Data that is included in the refinement applies to variables that are used in conditions of a loop statement. With help of a theorem prover and additional iterations for refinement the SLAM method tries to determine if the condition of a loop can cause the loop to terminate. In rare cases, it is possible that the theorem prover used by SLAM cannot solve the equations, which leads to a non-terminating algorithm. Consequently, verifying safety requirements becomes impossible. Our method does not use a theorem prover. If variables of a loop condition can change their values we assume that the condition eventually is violated, by which the loop terminates.

Counterexample-Guided Abstraction Refinement (CEGAR)[CGJ+03] is an automatic iterative abstraction-refinement methodology for which a datapath abstraction results in an approximation of the orignal design, i.e. if the approximation turns out to be too coarce, the approximation is automatically refined upto a point for which it can either generate a counter example or disprove it. While this technique is adaptive, our method is not. Therefore our approach can be seen as an instantiation of a first time right for CEGAR.

In D-Finder [BBNS09], a compositional method for checking invariance properties is presented. The basis of the method is an algorithm that iteratively computes invariants of components until these are strong enough to imply a global invariant that needs to be checked. In

contrast with our method, where an over-approximation of the model is obtained, the method used in D-finder over-approximates the local properties of the components.

Another approach related to ours, can be found in VeriSoft [God97] . Their approach consists of a systematic exploration of a state space by executing arbitrary code written in any language. They guarantee complete state space coverage up to some depth, hence a partial state space exploration. Consequently, this only guarantees safety properties up to a certain depth/bound and not for the entire system.

Another related approach is program slicing [Wei81]. This technique selects parts of the source code which are of interest to the values of specific variables. Our approach takes this to the extreme, by abstracting from all the variables and focus on calls between interfaces. Perhaps the technique of program slicing could also have been instrumental in abstracting from less relevant aspects of the model such as the logging of events.

7 Concluding remarks

In this paper, we have shown how safety properties can be verified for complex systems consisting of over more than 200 processes running in parallel. In particular, we have proposed a procedure for translating code specifying behaviour into mCRL2, with the help of an intermediate programming language preserving a simulation relation.

The smallest state space that we were able to derive from the model consisted of 76256 unique states and 253145 transitions for 236 tasks running in parallel. The rather small amount of states results from the dependencies between the mutual nodes.

In this paper, the method is described in terms of a general programming language SCPL and modelling language mCRL2. In principle, the method can be used in combination with many implementation languages (C, C++, C#, Pascal, Delphi, Java ...) and verification languages that have similar constructs for describing behaviour such as synchronized communication, sorts to encode different processes and non-determinism. The semantics-preserving translation of a real-life programming language to mCRL2 is considered future work.

Another possibility for future work is to add resources in the model e.g., time to complete a task, power consumption or the network traffic weight. By adding these it is possible to check whether the model exceeds maximal outer limits or claim more resources than available.

Future research can be conducted in the area of model refinement. If a safety property does not hold, it might be that the property is violated as a result of the over-approximation. The same holds for a liveness property that is fulfilled. To determine if these are artefacts the relevant conditions and variables need to be incorporated in the model. While this is feasible to do by hand for small systems, it is impossible for industrial sized systems. Therefore automatic model refinement, may be included in future research activities.

Finally, we are considering to study the over-approximation technique in isolation. For this approach, existing mCRL2 models that depend on data and for which the requirements are expressed independently of the data may be considered. On such models we may conduct experiments that indicate whether applying the abstraction technique from this paper significantly reduces the number of states and whether this helps in verifying requirements.

Acknowledgements This work is supported as part of the ITEA project Twins 05004. We would like to thank NBG Industrial Automation for the opportunity to conduct the experiments on the Lunaris, Muck van Weerdenburg for his assistance in obtaining the the temporal formulae, and the reviewers for their constructive suggestions and valuable contributions.

Bibliography

[AW02] T. Archer, A. Whitechapel. *Inside C#*. Pro-Developer Series. Microsoft Press, second edition, book & CD-rom edition, April 2002.

[BBNS09] S. Bensalem, M. Bozga, T.-H. Nguyen, J. Sifakis. D-Finder: A Tool for Compositional Deadlock Detection and Verification. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*. Pp. 614–619. Springer-Verlag, Berlin, Heidelberg, 2009.
http://dx.doi.org/10.1007/978-3-642-02658-4_45

[BR00] T. Ball, S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. Pp. 113–130. Springer-Verlag, London, UK, 2000.

[BR01] T. Ball, S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*. Pp. 103–122. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[Bra92] J. C. Bradfield. *Verifying Temporal Properties of Systems*. Progress in Theoretical Computer Science. Birkhäuser, 1992.

[CDH+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*. Pp. 439–448. 2000.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50(5):752–794, 2003.
http://doi.acm.org/10.1145/876638.876643

[CGP00] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. MIT Press, 2000.

[Dar86] I. F. Darwin. *Checking C programs with lint*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1986.

[Dja05] G. R. Djavanshir. Surveying the Risks and Benefits of IT Outsourcing. *IT Professional* 7(6):32–37, 2005.
http://dx.doi.org/10.1109/MITP.2005.153

[GG89] R. J. van Glabbeek, U. Goltz. Equivalence Notions for Concurrent Systems and Refinement of Actions (Extended Abstract). In Kreczmar and Mirkowska (eds.), *MFCS*. LNCS 379, pp. 237–248. Springer, 1989.

[GM99] J. F. Groote, R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In Haeberer (ed.), *Proc. Algebraic Methodology And Software Technology (AMAST 1998)*. LNCS 1548, pp. 74–90. Springer, 1999.

[GMR+07] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, M. van Weerdenburg. The Formal Specification Language mCRL2. In Brinksma et al. (eds.), *Methods for Modelling Software Systems (MMOSS)*. Dagstuhl Seminar Proceedings 06351. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2007.

[GMR+09] J. F. Groote, A. H. Mathijssen, M. A. Reniers, Y. S. Usenko, M. J. van Weerdenburg. Analysis of Distributed Systems with mCRL2. In Alexander and Gardner (eds.), *Process Algebra for Parallel and Distributed Processing*. Chapter 4, pp. 99–128. Taylor & Francis Group, 2009.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Pp. 174–186. ACM, New York, NY, USA, 1997. http://doi.acm.org/10.1145/263699.263717

[Gro] T. P. R. Group. QA-C++ toolsuite. `http://www.programmingresearch.com/`.

[Hol01] G. J. Holzmann. From Code to Models. In *ACSD '01: Proceedings of the Second International Conference on Application of Concurrency to System Design*. P. 3. IEEE Computer Society, Washington, DC, USA, 2001.

[Inc] P. Inc. PolySpace verification toolsuite. `http://www.polyspace.com`.

[Joh78] S. C. Johnson. a C Program Checker. In *Computer Science Technical Report*. Volume 65, pp. 78–1273. Murray Hill, 1978.

[Kof07] J. Kofron. Checking software component behavior using behavior protocols and spin. In Cho et al. (eds.), *SAC*. Pp. 1513–1517. ACM, 2007.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* 3(2):125–143, 1977.

[LME04] P. Leven, T. Mehler, S. Edelkamp. Directed Error Detection in C++ with the Assembly-Level Model Checker StEAM. In Graf and Mounier (eds.), *SPIN*. LNCS 2989, pp. 39–56. Springer, 2004.

[LNB+07] J. van Lier, I. V. Nieuwenhuyse, L. D. Boeck, T. Dohmen, N. Vandaele, M. Lambrecht. Benefits Management and Strategic Alignment in an IT Outsourcing Context. In *HICSS*. P. 206. IEEE Computer Society, 2007.

[MAG+99] S. Murthy, R. Akkiraju, R. Goodwin, P. Keskinocak, J. Rachlin, F. Wu, J. Yeh, R. Fuhrer, S. Kumaran, A. Aggarwal, M. Sturzenbecker, R. Jayaraman, R. Daigle, J. A. V. Mieghem. Coordinating Investment, Production, and Subcontracting. *Manage. Sci.* 45(7):954–971, 1999.

[NNH99] F. Nielson, H. R. Nielson, C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[RCK+92] C. Ramamoorthy, C. Chandra, H. Kim, Y. Shim, V. Vij. Systems integration: problems and approaches. *Systems Integration, 1992. ICSI '92., Proceedings of the Second International Conference on*, pp. 522–529, Jun 1992. 10.1109/ICSI.1992.217311

[Roo07] N. Roos. Océ geeft aanzet tot open innovatie in inkjet. June 2007. Mechatronica Magazine (Dutch). http://www.mechatronicamagazine.nl/nieuws/bekijk/artikel/oceacute-geeft-aanzet-tot-open-innovatie-in-inkjet.html

[SCK04] D. Siewiorek, R. Chillarege, Z. Kalbarczyk. Reflections on industry trends and experimental research in dependability. *IEEE Transactions on Dependable and Secure Computing* 1(2):109–127, April-June 2004. 10.1109/TDSC.2004.20

[ws08] J. P. web site. August 2008. `http://javapathfinder.sourceforge.net`.

[TDP03] O. Tkachuk, M. B. Dwyer, C. S. Pasareanu. Automated Environment Generation for Software Model Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*. Pp. 116–129. IEEE Computer Society, 2003.

[Web07] M. Weber. An Embeddable Virtual Machine for State Space Generation. In Bosnacki and Edelkamp (eds.), *Proceedings of the 14th International SPIN Workshop, Berlin, Germany*. Lecture Notes in Computer Science 2595, pp. 168–186. Springer Verlag, Berlin, 2007.

[Wei81] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*. Pp. 439–449. IEEE Press, Piscataway, NJ, USA, 1981.

Mobile CSP||B

Beeta Vajar, Steve Schneider and Helen Treharne

Department of Computing, University of Surrey, Guildford, Surrey, UK

Abstract: CSP||B is a combination of CSP and B in which CSP processes are used as control executives for B machines. This architecture enables a B machine and its controller to interact and communicate with each other while working in parallel. The architecture has focused on sequential CSP processes as dedicated controllers for B machines. This paper introduces Mobile CSP||B, a formal framework based on CSP||B which enables us to specify and verify concurrent systems with mobile architecture instead of the previous static architecture. In Mobile CSP||B, a parallel combination of CSP processes act as the control executive for the B machines and these B machines can be transferred between CSP processes during the system execution. The paper introduces the foundations of the approach, and illustrates the result with an example.

Keywords: CSP, B, mobility

1 Introduction

Numerous methods which combine state and event based models have been proposed: ZCCS [GS97], CSP-OZ [Fis97], Circus [OCW07], CSP2B [But00], ProB [LB03] and $CSP \parallel B$ [TS02, ST05]. Their advantage is that complex systems can be described and their verification ensures consistency of the models. Some integration, e.g., PIOZ [TDC04] and $\pi \mid B$ [KST07], support the description of mobility and dynamic patterns. This additional functionality is suitable for modelling agent systems or peer-to-peer networks where consideration of mobility is important. In this paper, we are interested in extending our $CSP \parallel B$ approach to include mobility and we have developed a formal framework so that we can compositionally verify the consistency of $CSP \parallel B$ specifications that include mobile aspects. This allows us to extend the range of specifications that we can write in $CSP \parallel B$ and retains our philosophy of not changing the underlying CSP [Sch00, Hoa85] and classical B [Sch01, Abr96] semantics. The framework adopts similar concepts to the architecture proposed in $\pi \mid B$. However, $\pi \mid B$ was limited to systems without inputs and outputs and was restricted to a framework to support divergence freedom verification. In this paper, we can deal with specifications that contain inputs and outputs and in addition to divergence freedom we can check for deadlock freedom. These two checks are the minimum verification that should be carried out in order to ensure that a mobile $CSP \parallel B$ specification is consistent. We use the divergence freedom check to confirm that B operations are called within their preconditions. In [Ros08] Roscoe introduces a new operator into a variant of CSP, introducing mobility in the way that the rights to use particular events are transferred between processes along special rights channels. Our approach is similar, in that channels can be transferred between processes. However, our work is motivated by the desire to retain access to the supporting

CSP and B toolsets, and so we aim to minimise the extension required to enable the form of mobility we aim to model.

1.1 The B-Method

The main unit of specification in the B-Method is an *abstract machine*. An abstract machine describes the state of the system in terms of mathematical structures such as sets, relations, functions and sequences. It also provides operations which change the state of the system. Each operation has a precondition or a guard. For the purposes of this paper we will restrict ourselves to preconditioned operations, as in classical B. Preconditioned operations have the form **PRE** P **THEN** S **END**, where P is the precondition, and S is the body of the operation, written in *Abstract Machine Notation (AMN)*, a simple language that contains assignment, choice, conditional, and precondition statements. A precondition expresses a predicate on the state of the machine which must hold when the operation is invoked, in order to ensure that the operation behaves as described by S; otherwise no guarantees can be given.

If S is an statement and Q is a predicate, the notation $[S]Q$ (also $wp(S,Q)$) denotes the weakest precondition which must be true when executing S to guarantee to reach a state in which Q is true.

The B-Method is a formal method supported by many comprehensive tools such as: B-Toolkit [BC02], ProB, and Atelier B [Cle09].

1.2 CSP

CSP is a theoretical notation or language for specifying and verifying concurrent systems, in terms of the events that they can perform. CSP provides a framework for describing and analysing interacting aspects of concurrent systems. Concurrent systems consist of interacting components known as processes. Each process works independently and may interact with its environment and other processes in the system. A process performs various events which describe its behaviour. CSP is an event-based formal language for designing and analysing a system behaviour through the events happening in the system. Its operators include event prefixing, channel input and output, choice, recursion, and parallel composition in which parallel components synchronise on events that they have in common. The variant of CSP that we will use in this paper is given in Section 3.

CSP has a variety of semantic models, based on observations. In this paper we are concerned primarily with traces and with divergences, though also with a need to handle deadlocks (which requires the failures model). A *trace* tr of a process P is a finite sequence of events which P is able to perform. The set of all possible traces of process P is denoted by $traces(P)$. A *divergence* of a process P is a sequence of events tr during or after which P can diverge—no guarantees can be made of its behaviour after divergence.

CSP is supported by highly efficient software tools such as ProBE [FSEL07b] and FDR [FSEL07a], supporting state exploration, refinement, divergence, and deadlock checking.

1.3 CSP||B

CSP || *B* is a parallel combination between CSP and B in which a CSP process is used as a control executive for a B machine. For each B machine's operation $bb \longleftarrow op(aa)$, there is a channel *op* between the CSP controller and the B machine which carries data types the same as the types of *aa* and *bb*. This provides the means for CSP controller and its controlled B machine to synchronise and communicate with each other while working in parallel. A B machine and its controller can send or receive values from each other through these channels. For instance, the CSP controller sends the value of *aa* to the B machine and receives the B machine's output, *bb*, through channel *op*. This means that in addition to control the execution order of the B operations, CSP controller and B machine can communicate with each other and they can exchange data and information through these channels while working in parallel in the system.

In [Mor90], Morgan introduces traces, failures and divergences semantics of CSP for action systems by using weakest precondition formulae. Based on this achievement, CSP semantics of traces, failures and divergences have been defined for B machines in [ST05]. Thus, a B machine can be understood as a CSP process. This common semantic framework makes it possible to define the parallel combination of B machines and CSP processes. In this framework the invocation of an operation outside its precondition corresponds to divergence.

According to the definitions in [Mor90], a trace of a B machine is a finite sequence of its operations. Divergence happens in a B machine when a pre-conditioned operation is called outside its precondition.

1.4 Introducing Mobility

In *CSP* || *B*, each CSP process can be the control executive of only one B machine and each B machine has only one CSP process as its controller. The architecture has focused on sequential CSP processes as dedicated controllers for B machines. The objective of this paper is to generalise *CSP* || *B* architecture in designing a new framework, *Mobile CSP* || *B*, which enables us to describe and verify systems in which a parallel combination of CSP processes are collectively the controllers of B machines, and each single B machine can be controlled by different CSP processes during the execution. By introducing mobility, each CSP process can receive a (mobile) machine or give it to another CSP process during the execution. An example of these kinds of systems is peer-to-peer networks in which data (B machines) can be transfered between the connected nodes (CSP controllers).

The following step is the consistency verification. We must ensure that B operations are always called within their preconditions, as they are passed between the controllers. We provide a theorem to establish divergence freedom of the whole mobile combined communicating system containing several CSP controllers each controlling several B machines, by establishing properties for each CSP controller separately. We also have the result that deadlock-freedom of the controllers implies deadlock-freedom of the combined system.

2 Mobile CSP || B

In standard CSP‖B, a controlled component consists of a CSP controller P in parallel with a B machine M. Operations op with inputs s and outputs t are declared in machines M as $t \longleftarrow op(s)$. In the combination they are treated as channels $op.s.t$. Standard CSP‖B has a static architecture in which one B machine works in parallel with only one CSP controller and each CSP controller can be the controller of only one B machine. So, the behaviour of the parallel combination is predictable as we have fixed controlled components during the system execution.

In Mobile CSP‖B, we intend to create a mobile architecture in which B machines are able to be transferred from one controller to another controller and each controller can work with more than one B machine at the same time. As controllers can exchange B machines between each other, B machines can have different controllers during their execution.

To enable machines to be passed around the system, we introduce a unique machine channel called *machine references*. CSP controllers use machine references as the link to interact with B machines. A machine reference is the only channel through which a CSP controller and a B machine can communicate with each other. As a result, a controller is only able to work with a machine if it owns that machine's reference. In other words, possession of a machine means having that machine's reference. In order for machines to be exchanged between controllers, machine references must be passed around between controllers in the system. Therefore, when a machine is going to be passed from one controller to another, the sender controller passes that B machine's reference to the other controller, as illustrated on the right in Figure 1. It shows that B machine M_1 is passed from CSP controller P_1 to P_2. The figure also shows the difference between Static CSP‖B architecture and Mobile CSP‖B architecture.

B machine M with machine reference z is presented in the system as $z : M$. All operations op in M are replaced with $z.op$. So, operation calls of the machine $z : M$ correspond to the communication $z.op.s.t$, and the machine reference z can itself be passed between controllers.

We introduce channels called *control points* between pairs of controllers on which machine references are passed around. When a machine is passed from one controller to another, the sender controller passes that B machine's reference to the other controller through the control point channel which exists between those two controllers.

We require that only one CSP controller is in possession of z at any one time, so that when z is passed from P_1 to P_2 then P_1 is no longer able to use z to call the operations of the machine. This will be the cornerstone for reasoning about the action of controllers on a mobile machine: that a controller has exclusive control over a machine it is using, and other controllers cannot interfere with its use of the machine.

We introduce MR as the set of machine references, CP as the set of control points, and C as the set of regular CSP channels. Each channel c in the set of regular channels C has a *type* denoted $type(c)$. The type of channels in CP is MR. Each machine reference in MR is associated with a particular B machine. The type of a machine reference z is the set of operations (with inputs and outputs) of the unique machine M that is associated with z.

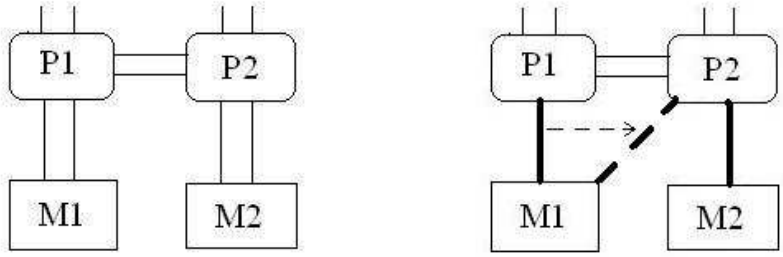


Figure 1: Static CSP||B architecture and Mobile CSP||B architecture

3 Mobile CSP Controllers

We will use the name *LOOP* to denote a mobile CSP controller. A process *LOOP* has a set of static *channels* $\chi(LOOP)$, which contains its communication channels and control points. Any particular control point in the alphabet of *LOOP* will be either incoming or outgoing with respect to *LOOP*, and is not permitted to be both. We identify the incoming control points within $\chi(LOOP)$ as $\chi_i(LOOP)$. The outgoing control points within $\chi(LOOP)$ are denoted $\chi_o(LOOP)$. The alphabet associated with communication channels is denoted $\chi_c(LOOP)$. These three sets are pairwise disjoint, and their union is $\chi(LOOP)$.

The syntax of mobile CSP controllers is defined by the following BNF:

P	$::=$	$SKIP \mid c?x \rightarrow P(x) \mid c!v \rightarrow P$	termination; communication
		$\mid cp_1?w \rightarrow P(w) \mid cp_2!z \rightarrow P \ \ (z \notin fv(P))$	passing machine references
		$\mid z.op!s?t \rightarrow P(t)$	operation call
		$\mid P' \,\Box\, P'' \mid P' \sqcap P'' \mid \mathit{if}\ b\ \mathit{then}\ P'\ \mathit{else}\ P''$	choice
		$\mid e \rightarrow P$	prefix
		$\mid N(v_1,\ldots,v_n)$	recursive call

where b is a boolean expression, e is an atomic CSP event which is not a B operation, $c \in \chi_c(LOOP)$, $cp_1 \in \chi_i(LOOP)$, $cp_2 \in \chi_o(LOOP)$, $v \in type(c)$, $z \in MR$, $t \longleftarrow op(s)$ is an operation of the B machine associated with z. $fv(P)$ is the set of free variables in P, including variables for machine references.

Sequential processes are then defined recursively as follows:

$$N_i(v_{i1},\ldots,v_{in}) \mathrel{\widehat{=}} P_i \qquad \text{where } fv(P_i) \subseteq \{v_{i1},\ldots,v_{in}\}$$

LOOP is then defined as one of the N_i.

The machine references that N_i knows initially will appear in the list $v_1,\ldots,v_n$. The set of machine references, *mr*, owned by *LOOP* can be defined by $mr(LOOP) = fv(LOOP) \cap MR$.

4 Parallel combination

A mobile combined communicating system including n controllers and m B machines is represented as

$$LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n \,||\, z_1 : M_1 \,||\, z_2 : M_2 \,||\, \ldots \,||\, z_m : M_m$$

where $z_1, z_2, \ldots z_m$ are the machine references for B machines $M_1, M_2, \ldots M_m$ respectively, and $i \neq j \Rightarrow z_i \neq z_j$

Mutual recursive CSP processes can be composed in parallel, only if (1) they have no machine references in common: $\forall 1 \leqslant i,j \leqslant n \bullet mr(LOOP_i) \cap mr(LOOP_j) = \varnothing$, (2) they differ on their incoming control points and their outgoing control points: $\forall 1 \leqslant j,k \leqslant n \bullet \chi_i(LOOP_j) \cap \chi_i(LOOP_k) = \varnothing$, $\forall 1 \leqslant j,k \leqslant n \bullet \chi_o(LOOP_j) \cap \chi_o(LOOP_k) = \varnothing$, and (3) each control point in the system has both a sender and a receiver. In other words, any outgoing (or incoming) control point in one controller is an incoming (or outgoing) control point of one of the other controllers in the system: $\bigcup_{j=1}^{n} \chi_i(LOOP_j) = \bigcup_{i=1}^{n} \chi_o(LOOP_i)$.

The free variables of the parallel combination of controllers is given as follows:

$$fv(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{i=1}^{n} fv(LOOP_i)$$

When a system is constructed, each machine reference must be given a different concrete value.

The alphabets for the parallel combination of controllers are given as follows:

$$\chi_i(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{j=1}^{n} \chi_i(LOOP_j)$$

$$\chi_o(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{i=1}^{n} \chi_o(LOOP_i)$$

$$\chi_c(LOOP_1 \,||\, LOOP_2 \,||\, \ldots \,||\, LOOP_n) = \bigcup_{i=1}^{n} \chi_c(LOOP_i)$$

The language of process terms and the rules for parallel combination of controllers have been designed to ensure that at any point in the system execution, only one controller has possession of any machine reference. Controllers do not share any machine references to begin with, and when a machine reference is passed along a control point to another controller, it is not retained by the sending controller.

In order to define the traces of parallel composition, it is necessary to keep track of the machine references as they are used and passed between controllers. We can define the projection of a trace onto a particular controller $LOOP$ given the channels $\chi_i(LOOP)$, $\chi_o(LOOP)$, $\chi_c(LOOP)$, provided we also know the set of machine references mr owned by the controller. This definition is based on the corresponding definition from [VST07].

The projection of a trace tr onto $\chi(LOOP)$ and a set of machine references mr can be defined inductively as shown in Figure 2 where $tr \upharpoonright \chi(LOOP), mr$ means the projection of tr onto alphabet of controller $LOOP$ who owns the set of machine references mr. This enables a definition of the traces of the parallel combination of controllers to be given:

$$traces(LOOP_1 \,||\, \ldots \,||\, LOOP_n) = \{tr \mid \forall 1 \leqslant i \leqslant n \bullet tr \upharpoonright \chi(LOOP_i), mr_i \in traces(LOOP_i)\}$$

$$
\begin{array}{rcl}
\langle\rangle \upharpoonright \chi(LOOP), mr & = & \langle\rangle \\
(\langle cp.z\rangle \frown tr) \upharpoonright \chi(LOOP), mr & = & \begin{cases} \langle cp.z\rangle \frown (tr \upharpoonright \chi(LOOP), mr \cup \{z\}) & \text{if } cp \in \chi_i(LOOP) \wedge z \notin mr \\ \langle cp.z\rangle \frown (tr \upharpoonright \chi(LOOP), mr - \{z\}) & \text{if } cp \in \chi_o(LOOP) \wedge z \in mr \\ tr \upharpoonright \chi(LOOP), mr & \text{if } cp \notin \chi_i(LOOP) \cup \chi_o(LOOP) \wedge z \notin mr \\ \text{undefined} & \text{otherwise} \end{cases} \\
(\langle c\rangle \frown tr) \upharpoonright \chi(LOOP), mr & = & \begin{cases} \langle c\rangle \frown (tr \upharpoonright \chi(LOOP), mr) & \text{if } c \in \chi_c(LOOP) \\ tr \upharpoonright \chi(LOOP), mr & \text{if } c \notin \chi_c(LOOP) \end{cases} \\
(\langle z.op\rangle \frown tr) \upharpoonright \chi(LOOP), mr & = & \begin{cases} \langle z.op\rangle \frown (tr \upharpoonright \chi(LOOP), mr) & \text{if } z \in mr \\ tr \upharpoonright \chi(LOOP), mr & \text{if } z \notin mr \end{cases}
\end{array}
$$

Figure 2: Projection of a trace onto $\chi(LOOP), mr$

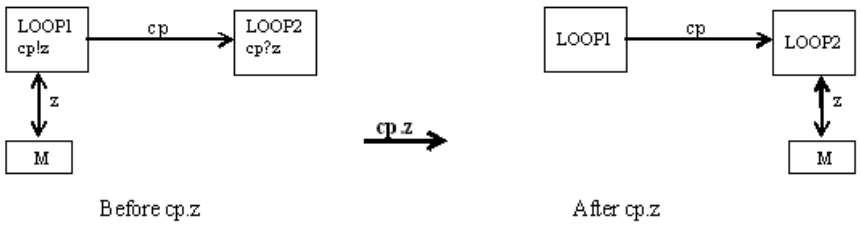


Figure 3: Transfer of machine M through control point cp

Whenever two controllers synchronise on *cp*, the machine reference is passed from one to the other, thus passing control over the associated machine. This is illustrated in Figure 3.

5 Consistency verification

In this section we discuss how divergence freedom of a mobile combined communicating system can be established. We also consider deadlock-freedom.

If the parallel combination of CSP controllers is not divergence free, then the whole system will have divergence. Therefore, the first step is to establish that the CSP part of the system is divergence free. FDR can be used to check divergence freedom of the CSP part of the system by checking the divergence freedom of the parallel combination of controllers. If the CSP part is divergence free, then any divergence in the system must arise from the B machines. Divergence arises in a machine when its operations are called outside their precondition by the controllers and we are essentially using the divergence freedom check to check this. Thus, the second step in divergence freedom verification is to establish that the operations of all machines in the system are always called inside their precondition by the controllers during the execution.

The key point is that we are allowing B machines to be passed from one controller to another. A controller typically receives a machine from another controller without knowing its state in advance, and so the divergence freedom between a machine and a controller needs to take the combined behaviour of the controllers into account. In order to keep proofs manageable, we need to check the state of the machines when passed from one controller to another, as the target controller does not have any control over the state of the received machine. Therefore, we will need to ensure that a machine is always transferred to another controller in a correct state where its operations will be called appropriately. In order to achieve this, for each control point we assign an assertion on the state of the machine whose reference is passed along that control point. The intention is that whenever a machine reference is passed to a CSP controller along a control point, it is guaranteed that the assertion is satisfied.

The notation $assert(cp_z)$ denotes the assertion of the control point cp for a machine whose reference is z. For instance, $assert(cp_z) : z.n = 0$ means that the variable n of the machine with machine reference z must be zero when passing through cp. For each control point, one assertion is assigned for all machines being passed through it. For a machine with machine reference w, the assertion of cp is $assert(cp_z)$ with w substituted for z which is $w.n = 0$.

To verify that a controller *LOOP* handles the B machines correctly, we translate the body of CSP processes N_i into AMN. We define a translation function, $trans(P_i)$, to translate the body of each process N_i into the corresponding AMN. Verifying $[trans(P)]Q$ will show that the sequence of operations expressed in P will establish the postcondition Q in the B machine, as used later in Definition 2.

Definition 1 *The translation of CSP expressions into AMN is defined as follows:*

$trans(SKIP) = SELECT\ false\ THEN\ skip\ END$

$trans(c?x \rightarrow P(x)) = ANY\ x\ WHERE\ x : type(c)\ THEN\ trans(P(x))\ END$

$trans(c!v \rightarrow P) = PRE\ v : type(c)\ THEN\ skip\ END;\ trans(P)$

$trans(cp?z \rightarrow P(z)) = ANY\ z\ WHERE\ z : MR\ THEN\ SELECT\ assert(cp_z)\ THEN\ trans(P(z))\ END\ END$

$trans(cp!z \rightarrow P) = PRE\ assert(cp_z)\ THEN\ skip\ END;\ trans(P)$

$trans(z.op!s?t \rightarrow P(t)) = t \longleftarrow z.op(s);\ trans(P(t))$

$trans(P' \,\square\, P'') = CHOICE\ trans(P')\ OR\ trans(P'')\ END$

$trans(P' \sqcap P'') = CHOICE\ trans(P')\ OR\ trans(P'')\ END$

$trans(if\ b\ then\ P'\ else\ P'') = IF\ b\ THEN\ trans(P')\ ELSE\ trans(P'')\ END$

$trans(e \rightarrow P) = skip;\ trans(P)$

$trans(N(v_1,\ldots,v_n)) = rec := N(v_1,\ldots,v_n)$

The last clause introduces a program counter *rec* to handle recursive calls. Observe that inputs $c?x$ and $cp?z$ are translated to the *ANY* statement, which models an assumption that the value being received is of the correct type. In the case of a machine reference, $cp?z$ also contains a *SELECT* statement, which models an extra assumption that the machine is in a state satisfying $assert(cp)$. Outputs $c!v$ and $cp!z$ are translated to *PRE* statement rather than *ANY* and *SELECT* statements. v and z are the parameters of the process so there are already some predicates on their value before this stage. Therefore, there is no need to have *ANY* in their translation. Instead, we

use the *PRE* statement, which models a guarantee that the condition is met on output values. In the case of a machine reference, there is also no *SELECT* statement in the translation of $cp!z$. This is because we are going to use weakest precondition formulae in our consistency verification strategy and the *PRE* statement is the suitable statement in order to detect when the assertion is not ensured by the sender process, which corresponds to divergence in the system.

Supposing for a process N_i in *LOOP* we can find an invariant referring to all free variables in N_i such that if this invariant is true then whenever N_i is called to be executed, it calls the operations of all the machines it owns at the beginning of that recursive call through their precondition. If we can establish that this invariant holds at every recursive call of N_i, and if the state of the machines N_i receives always satisfy the related assertions, then N_i always calls the operations of all the machines it works with through their precondition at all the time during the execution.

If we can establish the conditions above for each process N_i $(1 \leqslant i \leqslant n)$, then the parallel combination between *LOOP* and any machine it works with during the execution is ensured to be divergence free. As this invariant should be true at each recursive call, we call it Control Loop Invariant, CLI.

We now present the definition below for *LOOP* which contains the conditions we explained above:

Definition 2 *LOOP is called CLI preserver if for each* N_i $(1 \leqslant i \leqslant n)$ *in LOOP, a Control Loop Invariant,* CLI_i*, can be found such that :*

1. $[init_1;\ init_2;\ \ldots;\ init_m;\ rec := N_1](CLI_1)$
2. $\forall 1 \leqslant i \leqslant n \bullet ((rec = N_i \wedge CLI_i) \Rightarrow [trans(P_i)](\forall 1 \leqslant j \leqslant n \bullet (rec = N_j \Rightarrow CLI_j)))$

where $M_1,\ldots,M_m$ *are the machines that LOOP owns at the beginning of the execution and* $init_1,\ldots,init_m$ *are the Initialisation clause of machines* $M_1,\ldots,M_m$ *respectively.*

The theorem below makes use of this definition:

Theorem 1 *Supposing* $LOOP_1, LOOP_2, \ldots, LOOP_n$ *are the CSP controllers and* $M_1, M_2, \ldots, M_m$ *are the B machines in a mobile combined communicating system. If the parallel combination of controllers is divergence free and all controllers are CLI preserver, then the whole system,* $LOOP_1 \parallel LOOP_2 \parallel .... \parallel LOOP_n \parallel z_1 : M_1 \parallel z_2 : M_2 \parallel ... \parallel z_m : M_m$*, is divergence free.*

If each *LOOP* is a CLI preserver then this allows each *LOOP* to be separately checked for divergence-freedom on machines it controls at some point during its execution. This theorem allows all of these individual checks to be combined to an overall consistency result.

We can also establish deadlock-freedom of the overall system by checking deadlock-freedom of the combination of controllers. The B machines do not contribute to any deadlocking behaviour. This is because preconditions do not block, and we are not allowing blocking within the bodies of operations.

Theorem 2 *Suppose* $LOOP_1, LOOP_2, \ldots, LOOP_n$ *are the CSP controllers and* $M_1, M_2, \ldots, M_m$ *are the B machines in a mobile combined communicating system and the system is divergence*

free. If the parallel combination of controllers $LOOP_1 \parallel LOOP_2 \parallel .... \parallel LOOP_n$ *is deadlock free, then the whole system,* $LOOP_1 \parallel LOOP_2 \parallel .... \parallel LOOP_n \parallel z_1 : M_1 \parallel z_2 : M_2 \parallel ... \parallel z_m : M_m$*, is deadlock free.*

One important issue which has been considered in our work is to allow the refinements of the components into our framework. The intention is to be able to substitute a component by its refinement in such a way that the substitution does not have any effect on the system consistency properties. In [Vaj09], it has been proved that if we have a mobile combined communicating system and this system is divergence free and deadlock free, then if we use a refinement of B machines or CSP controllers instead of them in the system, the system remains divergence free and deadlock free. This enables us to use a refinement of a component instead of the component in the system.

6 Case study: Flight tickets sale system

In this section, we present a case study within Mobile $CSP \parallel B$ framework. The case study is a flight tickets sale system which presents the usage of Mobile $CSP \parallel B$ architecture in designing and developing peer to peer networks. We first provide a Mobile $CSP \parallel B$ model of a flight tickets sale system and then we verify the consistency of our model by using theorems 1 and 2. This is a simplified version of the case study to appear in [Vaj09]

This case study is designed as a flight tickets sale system in which tickets of different flights are sold or cancelled. The system contains a Sell agency which sells tickets of different flights to customers, and it contains one Return office which cancels customers' tickets. The Sell agency can only sell flight tickets and it is not able to cancel any tickets of the flights. The Return office is only responsible for cancelling tickets and it is not able to sell any flight tickets.

If the Sell agency or the Return office want to sell or cancel a ticket of a flight, they should have access to the information of that flight. Otherwise they are not able to sell or cancel any tickets. This description of the system makes it clear what should be modeled as the B machines and what should be modeled as the controllers in the system. The B machines in our system are the individual flights. They manage all the booking information of the flights. So each machine represents one of the flights in the system. The Sell agency and the Return office play the role of the controllers in our system.

Each flight machine is given a unique machine reference which is the channel used by the Sell agency and the Return office to contact and communicate with that flight machine. The Sell agency or the Return Office can sell or cancel a ticket of a flight only if they own that machine's reference. If they want to sell or cancel a ticket of a flight but they do not have that machine's reference, they request the machine's reference from each other.

The Sell agency and the Return office behave in such a way so that they do not keep the machines when they can not use them any more. A full machine can not be used any more by the Sell agency as all the tickets have already been sold and there is no more ticket available to be sold next. So, if a flight owned by a Sell agency is full after selling a ticket, the Sell agency passes that full machine to the Return office. On the other hand, an empty machine can not be used any more by the Return office as there is no sold ticket in the machine to be cancelled next.

So, if a flight owned by the Return office is empty after cancelling a ticket, the Return office passes that empty machine to the Sell agency. In other words, the Sell agency does not keep full machines with itself and the Return office does not keep empty machines with itself.

A set S is introduced for the Sell agency and for the Return office which contains all the machine references owned by the process. As a result, $SellAgency(S)$ represents the Sell agency which currently owns the machine references in S, and $ReturnOffice(S)$ represents the Return office which currently owns the machine references in S.

For the purposes of our case study, we will use two flight machines: *flight*1 and *flight*2. We also assume that at the beginning of the system execution, the Sell agency owns both flight machines. Therefore, the whole system is as below:

$$ReturnOffice(\{\}) \;||\; SellAgency(\{mr1, mr2\}) \;||\; mr1 : flight1 \;||\; mr2 : flight2$$

where $mr1$ and $mr2$ are the machine references of machines *flight*1 and *flight*2 respectively.

6.1 Design and specification of B machines

Each B machine manages the information of one flight in the system. The structure of all flights in our system are the same so the B machines have the same specification but with different names.

Each B machine contains the information about that particular flight such as: the (positive) number of seats of the flight, and the number of tickets which have already been sold. It also contains some operations for state transitions in the flight such as selling or cancelling tickets and some other operations for finding out the current state of the machine such as whether it is empty or full. Initially, each flight is empty. In other words, no ticket has been sold to anybody at the beginning of the execution. For reasons of space, only the operations of a flight machine are presented here, in Figure 4.

After specifying the flight machines in AMN, we used ProB to verify the internal consistency of our B machines and to explore the behaviour of their operations. As all B machines have the same structure in our system, a single B machine specification was checked, analysed and animated in ProB. The B machine was proved to be internally consistent and it behaved as expected.

6.2 System design and specification in mobile CSP

In this section, we describe the CSP specification of our system according to our mobile architecture. The Return office and the Sell agency are each specified as a CSP process which describes their behaviour in the system. In addition, it is defined who is the controller of each machine at the beginning of the execution.

A function *ref* is used to assign a unique machine reference for each machine in the system. By mapping each machine to a machine reference, the flight machines are given a unique machine reference which then can be used to communicate with the CSP processes. Two control points *dp* and *ep* are introduced for passing the machines between the Sell agency and the Return office. *dp* is a control point channel used to pass the flight machines from the Return office to the Sell agency. *ep* is a control point channel used to pass the flight machines from the Sell agency to the Return office.

```
response ← sell(pp) =
  PRE
    pp : Passport  &
    sold ≠ seats
  THEN
    IF pp : Passport − customer
    THEN
      customer := customer ∪ {pp}   ||
      sold := sold + 1   ||
      IF sold + 1 = seats
      THEN response := Full
      ELSE response := Available
      END
    ELSE response := IncorrectInput
    END
  END
```

```
response ← cancel(pp) =
  PRE
    pp : Passport  &
    sold ≠ 0
  THEN
    IF pp : customer
    THEN
      customer := customer − {pp}   ||
      sold := sold − 1   ||
      IF sold − 1 = 0
      THEN response := Empty
      ELSE response := Available
      END
    ELSE response := IncorrectInput
    END
  END
```

```
response ← empty =
  BEGIN
    IF sold = 0
    THEN response := YES
    ELSE response := NO
    END
  END
```

```
response ← full =
  BEGIN
    IF sold = seats
    THEN response := YES
    ELSE response := NO
    END
  END
```

Figure 4: Operations in a flight machine

The specification of the Sell agency and the Return office is shown in Figures 5 and 6 respectively.

A CSP process, *Controller*, is introduced which is the parallel combination of the Sell agency and the Return office. As we said before, we assume that at the beginning of the system execution, the Sell agency owns both flight machines. As a result, *Controller* is specified as: $Controller = ReturnOffice(\{\}) \;||\; SellAgency(\{mr1, mr2\})$.

The system can be coded for tool analysis into standard CSP, by treating the machine references as data values rather than as channels, and declaring a global channel *MC* (machine channel) which carries machine references as the first value, and then operation names and values as further values. In other words, any call of a machine operation $z.op!s?t$ in the body of the CSP controllers is modelled as $MC.z.op!s?t$ in the standard CSP description of our system.

After coding the system in standard CSP, the behaviour of the Sell agency and the Return office was checked individually by using ProBE. We then used ProBE to explore the execution of *Controller* in order to check their behaviour while working in parallel in the system. In addition, *Controller* was proved to be divergence free and deadlock free by using FDR.

6.3 Verification of the system: divergence-freedom

Our Flight tickets sale system will have divergence if (1) the parallel combination of the Sell agency and the Return office has divergence, or (2) a Sell agency calls the operation *sell* of a full machine during the execution, or (3) the Return office calls operation *cancel* of an empty machine during the execution.

$$SellAgency(S) = P_1(S)$$

$$P_1(S) = buy?flight?pn \rightarrow if\ ref(flight) \in S \\ then\ P_2(S, ref(flight), pn) \\ else\ P_3(S, ref(flight), pn) \\ \Box \\ ask?z \rightarrow P_4(S,z) \\ \Box \\ dp?z \rightarrow P_1(S \cup \{z\})$$

$$P_2(S,z,pn) = z.sell!pn?resp \rightarrow if\ resp = Full \\ then\ P_5(S,z) \\ else\ P_1(S)$$

$$P_3(S,z,pn) = (require.z \rightarrow ((dp?w \rightarrow P_2(S \cup \{w\}, w, pn)) \\ \Box \\ (fullMachine \rightarrow P_1(S)))) \\ \Box \\ (ask?w \rightarrow P_7(S,z,pn,w)) \\ \Box \\ (dp?w \rightarrow if\ w = z \\ then\ P_2(S \cup \{w\}, w, pn) \\ else\ P_3(S \cup \{w\}, z, pn))$$

$$P_4(S,z) = z.empty?resp \rightarrow \\ if\ resp = YES \\ then\ (emptyMachine \rightarrow P_1(S)) \\ else\ (ep!z \rightarrow P_1(S - \{z\}))$$

$$P_5(S,z) = (ep!z \rightarrow P_1(S - \{z\})) \\ \Box \\ (ask?w \rightarrow if\ w = z \\ then\ (ep!z \rightarrow P_1(S - \{z\})) \\ else\ P_6(S,z,w)) \\ \Box \\ (dp?w \rightarrow P_5(S \cup \{w\}, z))$$

$$P_6(S,z,w) = w.empty?resp \rightarrow \\ if\ resp = YES \\ then\ (emptyMachine \rightarrow P_5(S,z)) \\ else\ (ep!w \rightarrow P_5(S - \{w\}, z))$$

$$P_7(S,z,pn,w) = w.empty?resp \rightarrow \\ if\ resp = YES \\ then\ (emptyMachine \rightarrow P_3(S,z,pn)) \\ else\ (ep!w \rightarrow P_3(S - \{w\}, z, pn))$$

Figure 5: Sell agency

In this section we verify the divergence freedom of our system by using Theorem 1. *Controller* has already been proved to be divergence free by using FDR. The next step is to establish that the Sell agency and the Return office are CLI preserver. In order to achieve this, we should first assign assertions for control points in our system. Then, we should define Control Loop Invariants for the processes in the Sell agency and in the Return office.

If a machine is passed from the Sell agency to the Return office, it should not be empty. On the other hand, if a machine is passed from the Return office to the Sell agency, it should not be already full. *ep* is the control point which passes the machines from the Sell agency to the Return office. So, the assertion of ep should be assigned as $assert(ep_z) : z.sold \neq 0$. *dp* is the control point which passes the machines from the Return office to the Sell agency. So, the assertion of *dp* should be assigned as $assert(dp_z) : z.sold \neq z.seats$

For the Sell agency and the Return office, we can introduce Control Loop Invariants, shown in Figure 7 which establish that both Sell agency and the Return office are CLI preserver. Thus, according to Theorem 1 our system is divergence free.

6.4 Verification of the system: deadlock-freedom

By verifying deadlock freedom of our Flight tickets sale system, we establish that there will never occur a situation in which the execution of our system is blocked. This is a natural condition checked for concurrent systems.

The deadlock freedom of our system is verified by using Theorem 2. We have already proved

$$ReturnOffice(S) = R1(S)$$

$$\begin{aligned}
R_1(S) = {} & return?flight?pn \rightarrow if\ ref(flight) \in S \\
& \quad then\ R_2(S, ref(flight), pn) \\
& \quad else\ R_3(S, ref(flight), pn) \\
& \Box \\
& require?z \rightarrow R5(S,z) \\
& \Box \\
& ep?z \rightarrow R_1(S \cup \{z\})
\end{aligned}$$

$$\begin{aligned}
R_2(S,z,pn) = {} & z.cancel!pn?resp \rightarrow if\ resp = Empty \\
& \quad then\ R_4(S,z) \\
& \quad else\ R_1(S)
\end{aligned}$$

$$\begin{aligned}
R_3(S,z,pn) = {} & ask!z \rightarrow (ep?w \rightarrow R2(S \cup \{w\}, w, pn) \\
& \quad \Box \\
& \quad emptyMachine \rightarrow R_1(S)) \\
& \Box \\
& require?w \rightarrow R_7(S,z,pn,w) \\
& \Box \\
& ep?w \rightarrow if\ w = z \\
& \quad then\ R_2(S \cup \{w\}, w, pn) \\
& \quad else\ R_3(S \cup \{w\}, z, pn)
\end{aligned}$$

$$\begin{aligned}
R_4(S,z) = {} & dp!z \rightarrow R_1(S - \{z\}) \\
& \Box \\
& require?w \rightarrow if\ w = z \\
& \quad then\ dp!z \rightarrow R_1(S - \{z\}) \\
& \quad else\ R_6(S,z,w) \\
& \Box \\
& ep?w \rightarrow R_4(S \cup \{w\}, z)
\end{aligned}$$

$$\begin{aligned}
R_5(S,z) = {} & z.full?resp \rightarrow \\
& \quad if\ resp = YES \\
& \quad then\ fullMachine \rightarrow R_1(S) \\
& \quad else\ dp!z \rightarrow R_1(S - \{z\})
\end{aligned}$$

$$\begin{aligned}
R_6(S,z,w) = {} & w.full?resp \rightarrow \\
& \quad if\ resp = YES \\
& \quad then\ fullMachine \rightarrow R_4(S,z) \\
& \quad else\ dp!w \rightarrow R_4(S - \{w\}, z)
\end{aligned}$$

$$\begin{aligned}
R_7(S,z,pn,w) = {} & w.full?resp \rightarrow \\
& \quad if\ resp = YES \\
& \quad then\ fullMachine \rightarrow R_3(S,z,pn) \\
& \quad else\ dp!w \rightarrow R_3(S - \{w\}, z, pn)
\end{aligned}$$

Figure 6: Return office

in previous section that the system is divergence free. Finally, *Controller* has already been proved to be deadlock free by using FDR. All conditions in Theorem 2 are true. Thus, our system is deadlock free.

7 Conclusion

In this paper, we introduced Mobile $CSP \parallel B$, a formal framework based on $CSP \parallel B$ which enables us to specify and verify concurrent systems with mobile architecture instead of the previous static architecture. In the previous static version of $CSP \parallel B$, for each operation in a B machine, there is one channel between that machine and its controller. However in our work, a machine's reference is the only channel through which a CSP controller and that B machine can interact with each other. In contrast to static $CSP \parallel B$ in which each controller is dedicated for one B machine, we designed our framework in such a way that controllers are able to work with different machines during the execution. Controllers exchange machines between each other by exchanging the machine references. This results from two facts about our system architecture:

1. In Mobile $CSP \parallel B$, we have introduced mobile channels: machine references are mobile channels and they can move around the system during the execution.

2. In Mobile $CSP \parallel B$, mobile channels (machine references) are allowed to be passed between processes through static channels (control points) in the system.

$CLI_{P_1(S)} : S \subseteq MR \wedge \forall k \in S \bullet k.sold \neq k.seats$

$CLI_{P_2(S,z,pn)} : CLI_{P_1(S)} \wedge z \in S \wedge pn \in Passport$

$CLI_{P_3(S,z,pn)} : CLI_{P_1(S)} \wedge z \in (MR - S) \wedge pn \in Passport$

$CLI_{P_4(s,z)} : CLI_{P_1(s)} \wedge z \in S$

$CLI_{P_5(S,z)} : S \subseteq MR \wedge \forall k \in (S - \{z\}) \bullet k.sold \neq k.seats \wedge z \in S \wedge z.sold \neq 0$

$CLI_{P_6(S,z,w)} : CLI_{P_5(S,z)} \wedge w \in S \wedge z \neq w$

$CLI_{P_7(S,z,pn,w)} : CLI_{P_3(S,z,pn)} \wedge w \in S$

$CLI_{R_1(S)} : S \subseteq MR \wedge \forall k \in S \bullet k.sold \neq 0$

$CLI_{R_2(S,z,pn)} : CLI_{R_1(S)} \wedge z \in S \wedge pn \in Passport$

$CLI_{R_3(S,z,pn)} : CLI_{R_1(S)} \wedge z \in (MR - S) \wedge pn \in Passport$

$CLI_{R_4(S,z)} : S \subseteq MR \wedge \forall k \in (S - \{z\}) \bullet k.sold \neq 0 \wedge z \in S \wedge z.sold = 0$

$CLI_{R_5(S,z)} : CLI_{R_1(S)} \wedge z \in S$

$CLI_{R_6(S,z,w)} : CLI_{R_4(S,z)} \wedge w \in S \wedge w \neq z$

$CLI_{R_7(S,z,pn,w)} : CLI_{R_3(S,z,pn)} \wedge w \in S$

Figure 7: Control Loop Invariants in the Sell agency and the Return office

In addition, we defined and verified the conditions which guarantee the divergence freedom and deadlock freedom consistency of the systems specified and designed in Mobile *CSP* || *B*.

The case study demonstrates the applicability of our mobile *CSP* || *B* framework in specifying and verifying mobile communication systems. It shows that the theorems are sufficient to manage concurrent updates of state. It also demonstrates the ability of the CSP controllers to interact with numerous machines at the same time.

Apart from formal method integrations, some other approaches have also been created in modelling and verifying mobile systems one of which is Mobile UNITY. Mobile UNITY [MR96], an extension of the parallel program design language UNITY [CM88], is a language and proof logic for specifying and reasoning about concurrent mobile systems. In Mobile UNITY, components can move around and execute at different locations and they can interact and communicate with each other during the execution. In Mobile UNITY notation, mobility is modelled as the change of the location of components. In other words, the change in the location of components provides means to model movement in the system. It allows the description of location-sensitive behaviour, e.g., interaction at the same place, or within a certain distance. Each component in Mobile UNITY has a distinguished location variable. The location of each component is modelled by assignment of a value to its location variable. The movement of a component is modelled by the change of value of its location variable. Mobile UNITY proof logic is employed to verify the safety and liveness properties of a system expressed in the Mobile UNITY notation. Mobile UNITY has no notion of refinement.

We believe using CSP language in our approach makes flow of control more explicit in contrast to approaches that use control variables. In addition, Mobile *CSP* || *B* framework supports a refinement approach: it enables the refinements of components to be a substitute of the original components in the system while the system consistency is guaranteed to remain. Furthermore, as our framework can be coded into the original constructs of CSP and B, we are able to use the comprehensive and highly efficient software tools of CSP and the B-Method to analyse, animate and check the behaviour and the consistency of the two parts of a mobile system separately. This shows the advantage of using B-Method as our chosen state based formal method and CSP as the process algebra in our framework.

The approach taken in $\pi \mid B$ [KST07] also allows dynamic creation of machines and channels, and reconfiguration of the network. In contrast, the approach taken in this paper provides a more static network between the controllers, but developing the framework to enable reconfiguration, and dynamic process and machine creation, would be an interesting avenue of future research.

Acknowledgements: We are grateful to the anonymous reviewers for their comments.

Bibliography

[Abr96] J. R. Abrial. *The B Book: Assigning Programs to Meaning*. CUP, 1996.

[BC02] B-Core. B-Toolkit. 2002.
http://www.b-core.com/btoolkit.html

[But00] M. Butler. csp2B: A Practical Approach To Combining CSP and B. *Formal Aspects of Computing* 12, 2000.

[Cle09] ClearSy. Atelier B 4.0. 2009.
http://www.atelierb.eu/index-en.php

[CM88] K. M. Chandy, J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[FSEL07a] Formal Systems (Europe) Ltd. FDR 2.83 Manual. 2007.
http://www.fsel.com

[FSEL07b] Formal Systems (Europe) Ltd. ProBE. 2007.
http://www.fsel.com

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In *FMOODS '97*. 1997.

[GS97] A. J. Galloway, W. J. Stoddart. An operational semantics for ZCCS. *ICFEM '97*, 1997.

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

[KST07] D. Karkinsky, S. Schneider, H. Treharne. Combining mobility with state. *IFM'07*, 2007.

[LB03] M. Leuschel, M. Butler. ProB: A Model Checker for B. In *FM 2003*. 2003.
http://www.stups.uni-duesseldorf.de/ProB/overview.php

[Mor90] C. C. Morgan. Of wp and CSP. *In W.H.J. Feijen, A. J. M. van Gesteren, D. Gries, and J. Misra, editors, Beauty is our business: a birthday salute to Edsger W. Dijkstra. Springer-Verlag*, 1990.

[MR96] P. J. McCann, G. C. Roman. Mobile UNITY: A language and logic for concurrent mobile systems. *Technical Report WUCS-97-01, Department of Computer Science, Washington University in St. Louis*, 1996.

[OCW07] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing* 21, 2007.

[Ros08] A. Roscoe. On the expressiveness of CSP. 2008. Draft of October 23, 2008.

[Sch00] S. Schneider. *Concurrent and Real-time Systems: the CSP approach*. Wiley, 2000.

[Sch01] S. Schneider. *The B-method: an introduction*. Palgrave Macmillan, 2001.

[ST05] S. Schneider, H. Treharne. CSP Theorems for Communicating B machines. *Formal Aspects of Computing* 17, 2005.

[TDC04] K. Taguchi, J. Dong, G. Ciobanu. Relating pi-calculus to Object-Z. *ICECCS*, 2004.

[TS02] H. Treharne, S. Schneider. Communicating B machines. *ZB2002*, 2002.

[Vaj09] B. Vajar. *Mobile CSP||B*. PhD thesis, University of Surrey, in preparation, 2009.

[VST07] B. Vajar, S. Schneider, H. Treharne. Introducing Mobility into CSP||B. In *AVOCS 2007*. 2007.

Short Submissions

B Based Verification for Real Time Systems

Anaheed Ayoub[1], Ayman Wahba[2], Ashraf Salem[1] and Mohamed Sheirah[2]

[1] Mentor Graphics Egypt
[2] Ain Shams University

Abstract: This paper presents an approach for modelling and verifying real time systems using B. We first model the real-time systems using a network of timed automata and write the system-required properties using TCTL formulas. And then represent this model using the B language that enables us to take full advantages of the B-method features and associated tools.

Keywords: B-method, Real-time systems, timed automata, TCTL

1 Introduction

The motivation of our work is the search of efficient specification, design and verification methods for real-time systems. We do so by integrating the timed automata [AD94], TCTL formulas [ACD93] and the B-method [Abr96]. We present methods to automate the derivation of B-model from network of timed automata plus TCTL formulas to be able to take advantages of the B-method features including the refinement mechanism. In order to do so, the entire system needs to be represented using B language. We create an automatic mapper to do that.

The main idea of representing the timed automata as B-model was introduced in our previous work [AWS08]. In this paper we extend this work by mapping the TCTL properties into B, and by applying our mechanism to real test cases and verifying the correctness of our approach using these test cases. The modelling of real time properties in B has been addressed in [RBFN05]. The main step forward of our work is that (1) our work models the entire system with its timing properties, taking the synchronization between the system components into account (2) Our approach is generic and it is totally automatic (3) also the generated B using our approach can be verified using the standard B tools.

A brief description for our approach to represent the timed automata model plus TCTL properties using B is given in section 2. In section 3, we represent the experimental results obtained using this approach. Finally our conclusions are given in section 4.

2 Representing Timed Automata and TCTL Formulas in B

Figure 1 shows the overall flow of the mapper form system of network of timed automata with its TCTL properties into B-model. The input to the overall flow is .xml and .q files of the timed automata model and the TCTL properties respectively. The first step is the parsing, the output of this step is our timed automata and TCTL object structures, then these object structures are used as an input to the last step which is the writing one, the overall output is B-Method .mch files that represent the input system.

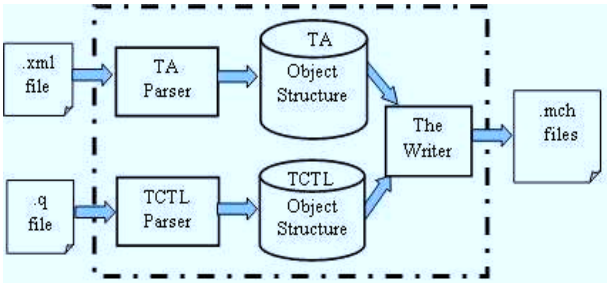


Figure 1: The Overall Flow

About our representation for the timing aspects, timed automata use continuous time domain, while B-method supports discrete variables only, so we represent the timed automata clocks using B integer variables that can be seen as discrete version of these continuous clocks. This is the same suggestion used at [RBFN05]. This discrete model is sufficient for our representation as the timed properties need to be hold within fixed time bound. To increase the clocks synchronously with the same rate, we create a separate B operation named "clock_increment" to increment the clocks by one time unit.

Each generated B-machine is written into different .mch file. And about the writing of each object, it depends on the object type on which we will decide what exactly are needed to be written. Note that both the timed automata and the TCTL formulas are represented as a single B-model, as the TCTL formulas are just some properties of this system, and so they are basically represented as a part of the B-model invariant; the invariant of the root machine.

The approach that we used to check the correctness of the transformation is by showing that the verified TCTL properties against the timed automata are kept satisfied for the corresponding B-model. It is not the full correctness verification but at least it allows us to verify that the target specification preserves those properties that were of particular interest in the source specification.

3 Experimental Results

We have applied the suggested approach to the test cases summarized in Table 1. The references that contain these test cases details can be found at [ben09] while the details of the last testcase (PC) can be found at [AWSS03]. The TCTL properties of the timed automata models of these systems are found to be verified against the generated B-model too. We used AtelierB[ate09] and ProB[pro09] tools for this verification.

4 Conclusions

In this paper, we have presented an approach to model and verify real time systems in B. this approach is build on the integration of the timed automata, TCTL formulas and B-method. We automatically drive B-model from network of timed automata plus its TCTL properties to be able to take advantages of the B-method features including the refinement mechanism, and the

Table 1: Testcases descriptions

	Description
Bridge	2 soldiers and single torch, while each soldier has a different speed
Fischer	A mutual exclusion algorithm
CSMA /CD	Carrier Sense, Multiple-Access with Collision Detection, which is a protocol describes one solution to the problem of assigning a multi-access bus to one of many stations arises
FDDI	Fiber Distributed Data Interface, which is a fiber-optic token ring local area network
BOCDP	Bang & Olufsen Collision Detection Protocol, which is applied for the exchange of control information between audio/video units in the company's product line
PC	The production cell is an industrial process that is used to forge metal plates in a press.

B associated tools. A tool that implements our approach has been created. This tool automatically generates B-model from network of timed automata system plus TCTL formulas. We have verified the applicability of our approach using this tool against a couple of test cases.

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ACD93] R. Alur, C. Courcoubetis, D. L. Dill. Model-Checking in Dense Real-Time. *In Journal of Information and Computation*, 1993.

[AD94] R. Alur, D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science* 126(2):183–235, 1994.

[ate09] AtelierB Manual. www.it.uu.se/edu/course/homepage/bkp/vt08/AB/bbin/help/MUtdm.html, 2009.

[AWS08] A. Ayoub, A. Wahba, M. Sheirah. Mapping Timed Automata to B. In *Proc. of International Design and Test Workshop, IDT'08*. Pp. 255–259. Monastir, Tunisia, 2008.

[AWSS03] A. Ayoub, A. Wahba, A. Salem, M. Sheirah. TCTL-Based Verification of Industrial Processes. In *Proc. of FORUM on Specification and Design Languages, FDL'03*. Pp. 456–468. Frankfurt, Germany, 2003.

[ben09] Test Cases. http://www.it.uu.se/research/group/darts/uppaal/benchmarks, 2009.

[pro09] ProB Manual. http://www.stups.uni-duesseldorf.de/ProB/ProB_User_Manual.pdf, 2009.

[RBFN05] M. Rached, J.-P. Bodeveix, M. Filali, O. Nasr. Real Time Aspects: Specification and Composition in B. In *Proc. of 7th International Workshop on Aspect-Oriented Modeling, MODELS'05*. 2005.

Learning from experts to aid the automation of proof search

Alan Bundy[1], **Gudmund Grov**[2] **and Cliff B. Jones**[3]

[1] University of Edinburgh [2] Heriot-Watt University [3] University of Newcastle

Most formal methods give rise to proof obligations which are putative lemmas that need proof. Discharging these POs can become a bottleneck in the use of formal methods in practical applications. Some techniques for reducing this bottleneck are known — it is our aim to increase the repertoire of techniques by tackling learning from proof attempts. Even after obvious fixed heuristics are used, there remains the problem of what to do with POs that are *not* discharged automatically. In many cases where a correct PO has not been discharged, an expert can easily see how to complete a proof. We believe that it would be acceptable to rely on such expert intervention to do one proof if this would enable a system to kill off others "of the same form".[1]

Our objective is to significantly decrease the human effort in doing top-down formal development. We believe this can be achieved by increasing the proportion of POs that are discharged by the system.

Without questioning the value of any other approach to improving software development, we wish to be precise about the setting to which we hope to contribute. We are interested in helping engineers discharge POs that arise in the development process from a (formal) specification to a completed design.

Experience also tells us that change is of the essence and the impact of this gives another insight into the advantages of learning. When a change is made, a user has to redo proofs. The Rodin Toolset is already good at tracing the impact of changes and reducing the proof rework. But it would be a bonus if the system could learn enough from hand proof attempts at pre-change POs to discharge automatically similar proof obligations after a change. Of course, there will always be differences that defeat this strategy.

The main hypothesis to be addressed is: *enough information can be automatically extracted from a hand proof that examples of the same class can be proved automatically*.

We believe that it is possible to build a system that will learn enough from one proof attempt to significantly improve the chances of proving "similar" results automatically. By "proof attempt" we include things like the order of the steps explored by the user (not just the finished chain in the final proof). Thus it is central to our goal that we find *high-level* strategies capable of cutting down the search space in proofs. What we are looking for is at a much higher level than LCF-style tactic languages — such tactics are programs to construct proofs and are brittle in the sense that they behave differently for similar POs.

We believe that by separating information about data structures and approaches to different patterns of POs, a taxonomy begins to evolve. A PO approach might be seen to use "generalise induction hypothesis" in a specific proof about, say, sequences; a future use of this PO might involve a more complicated tree data structure — but if it has an extended induction rule (e.g. by adding an argument to accumulate values), the same strategy might work.

[1] To see how useful such a facility could be, note that at the first review of DEPLOY, one of the industrial partners reported an application that gave rise to some 300 POs; of these, about 200 were discharged automatically; five really difficult proofs were done by hand; although the remaining 95 "followed the pattern" of the five, they also had to be done slowly and manually.

So our hypothesis can be recast as: *we believe that it is possible to (devise a high-level strategy language for proofs and) extract strategies from successful proofs that will facilitate automatic proofs of related POs.*

Designing the strategy language is a part of a proposed project but it might be useful to give some indications of what we expect it to look like. Our strategy language will combine a high-level proof strategy with a "vocabulary" of terms that might be instantiated in the separate theories of data structures stored in the system. The meta-language employed in our rippling/induction proof-planning work provides an existence proof for such a strategy language. Items that we expect to play a major part include:

- *Some 'standard' proof plans and known deviations from and patches to them.* [BBHI05] uses rippling to describe a 'standard' proof plan for inductive proofs and shows how each different pattern of failure in rippling suggests a different way of patching a failed proof attempt. We hypothesise that expert-provided proofs of undischarged POs will typically exhibit either a new proof plan or a new patch to an existing plan.

- *Choices of unusual induction rules and variables, choices of loop invariants.* Choosing an alternative non-standard induction rule is one of the patches to the standard induction proof plan which is described in [BBHI05]; patching a failed loop invariant is the patch described in [SI98].

- *Choices of intermediate lemmas.* Designing, constructing and proving a key intermediate lemma is one of the patches to the standard induction proof plan described in [BBHI05].

- *Generalisation of the PO.* [BBHI05] also describes a couple of ways of generalising the PO or the current goal to patch a failed proof.

Lemmas, case splits, loop invariants, generalisations and their points of application all need to be described in an abstract form if they are to apply to all members of a family of proofs. This is because the details will vary from proof to proof, but there may be a level of abstraction at which their descriptions coincide. Rippling, for instance, provides an exemplar abstract language, since it can describe 'missing' intermediate lemmas in terms of subexpressions that must match with different parts of the current goal. We will also make use of generic taxonomies, for instance, types of induction rule, types of generalisation, etc. to support the abstraction of proofs and their subsequent application to new conjectures. For instance, the use of a two-step induction on a recursive data-structure in the source proof must first be abstracted in order to be applied to a different data-structure in the target proof. We expect to develop and use additional kinds of abstraction during the course of the project.

The major challenge is to design a sufficiently general-purpose and robust strategy language so that it can deal with unanticipated proof plans and patches that experts will devise. If we knew in advance what these plans and patches would be, we could include them in the theorem prover, so that the problematic POs would be discharged and would not require expert attention.

Bibliography

[BBHI05] A. Bundy, D. Basin, D. Hutter, A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge Tracts in Theoretical Computer Science 56. Cambridge University Press, 2005.

[SI98] J. Stark, A. Ireland. Invariant Discovery via Failed Proof Attempts. In Flener (ed.), *Logic-based Program Synthesis and Transformation*. LNCS 1559, pp. 271–288. Springer-Verlag, 1998.

Specifying Railway Interlocking Systems*

Karim Kanso and Anton Setzer

Swansea University, Wales, UK

Abstract: One of the Grand Challenges in Computer Science is to verify railway interlocking systems [1]. We give a generic datatype of control tables and ladder logic (2,3), and extract from these verification conditions (4). A proof of the correctness of these conditions is performed using induction and a datatype of reachable states (5). Finally, some concluding remarks are presented (6). This specification has been implemented in Agda2.

Keywords: verification, specification, dependent types, Agda2, railway, control system, interlocking, ladder logic

Previously, [3] we developed a verification system that applied SAT solver technology to verify arbitrary first order formulæ w.r.t. safety. This work builds upon previous work by formalising what a control table is and what it means for an interlocking system to ratify a verification condition.

1 Physical Layout

Initially, when designing a railway, it is required to fix a physical layout of the involved hardware, i.e. track segments, signals, platforms, emergency systems, etc. These pieces of hardware have attributes, i.e. track segments can be occupied or unoccupied and signals can be green or red. Many techniques have been used to formally model the physical layout of a railway [4]. Layouts must contain all identifiers of the components along with their relationships; they are viewed as a signature for the interlocking. Let *Layout* be the type of physical layouts, and $p : Layout$.

2 Control Tables

Control tables are used to define the behaviour of the hardware in traditional railway signalling systems; they are a set of rules/constraints that must be observed and are an abstract specification for a portion of the railway. Responsibility of the safety of the railway is delegated to control tables, which express for instance that a green signal is only shown when a given constraint is met.

Control tables are sentences built over a physical layout, in our system we define a control

* This research is funded by Westinghouse Rail Systems, Chippenham, UK

table to be a list of relations, which formalise the constraints. Relevant signatures are:

$$\begin{array}{l}\text{data } ControlTableEntry_p \text{ where}\\ \quad route : RouteID \to Signal \to Signal \to List(Track) \to ControlTableEntry_p\\ \quad < \text{other relations} >\end{array}$$

$$ControlTable_p = List\ ControlTableEntry_p$$

An entry $route(r, s_0, s_1, [t_1, \ldots, t_n])$ expresses that route r starts at signal s_0, ends at signal s_1, and uses track segments $t_1, \ldots, t_n$.

3 Ladder Logic

Interlocking systems are realised using a multitude of techniques; systems programmed using ladder logic are the focus of this research. Ladder logic is a discrete time, linear system of Boolean equations, relating Boolean valued inputs and internal state to Boolean valued outputs. Ladder logic programs are represented using a transition function $next : Internal \times Input \to Internal$ between states, an output function $output : Internal \times Input \to Output$ and an initial internal state. In the following, let $Ladder$ be the type of ladder logic programs, and $l : Ladder$; $Internal$, $Input$ and $Output$ are indexed by l. Notably, l is a model of a control table c *iff* l never violates the constraints in c.

4 Verification Conditions

To determine whether a ladder logic program correctly refines a control table, verification is required. Firstly, the datatype of correctness is defined as a relation between the input and output of the system; $Correctness_l \subseteq Input_l \times Output_l$. Secondly, the verification condition type is defined as a function from a control table entry to $Correctness_l$.

$$VerifCond^l_p = ControlTableEntry_p \to Correctness_l$$

An intersection between multiple correctness relations is used to construct combined correctness relations. This technique can be used to verify all the control table.

Verification conditions are sentences built over a physical layout; they can be generated from the relations in the control table by instantiating a template sentence. E.g. for a signal with only one route, such as exist in some installations the *route* relation can be mapped to a correctness relation which states "if a track segment in the route is occupied, then the first signal shows a red aspect".

$$\begin{array}{l}f : VerifCond^l_p\\ f(route(rt, s_1, s_2, ts))(in, out) \Leftrightarrow\\ \quad fold(\vee, false, map((\lambda x \bullet in(x.occupied)), ts)) \to out(s_1.red)\end{array}$$

Correctness conditions can be independent from the control table entries; this is particularly useful when verifying general safety properties. E.g. a signal does not display both red and green aspects at the same time[1].

[1] In practice the signalling policy in use could allow for a transitional period where both aspects are shown.

5 Correctness

Verification uses the principle of induction, working from an initial internal state at time 0 up to time n. States might be unreachable causing false negatives during the verification. This problem is avoided by using a datatype of reachable states.

Reachable states are defined using induction-recursion [2]. The initial state is reachable by definition; from any reachable state at time t, after processing the inputs, state $t+1$ is also reachable. We obtain a type $ReachableState_l$ and a function $toInternal_l : ReachableState_l \to Internal_l$. A proof of correctness is then a proof that the relation is not empty, i.e.

$$correct_l^q : (r : ReachableState_l) \to (i : Input_l) \to q(i, output_l(toInternal_l(r), i))$$

where $q : Correctness_l$.

6 Concluding Remarks & Future Work

The above scheme has been implemented using the theorem prover and programming language Agda2[2], which is based on constructive type theory (intuitionistic logic). We have verified simple toy examples; to verify complex problems we intend to explore the possibility of linking Agda2 to a SAT solver. Also, a CASL implementation is planned.

We conjecture that an institution can be defined where the signatures are physical layouts (plus a chosen logic signature), sentences are control tables (moreover a sentence is a formula built using the chosen logic), models are ladder logic programs and the satisfaction relation is derived from the correctness proof.

Bibliography

[1] Bjørner, D., *TRain: The Railway Domain*, in: *Building the Information Society*, IFIP International Federation for Information Processing **156/2004** (2004), pp. 607–611.
URL http://www.springerlink.com/content/527p7237102w5741/

[2] Dybjer, P. and Setzer, A., *Indexed Induction-Recursion*, in: *Proof Theory in Computer Science*, LNCS Lecture Notes in Computer Science **2183** (2001), pp. 93–113.
URL http://www.springerlink.com/content/rc3t3m7gkfnmpq3d/

[3] Kanso, K. and Moller, F. and Setzer, A., *Automated Verification of Signalling Principles in Railway Interlocking Systems*, to appear in: ENTCS, (2009)

[4] Pnika, M., *Formal Approach to Railway Applications*, in: *Formal Methods and Hybrid Real-Time Systems*, LNCS Lecture Notes in Computer Science **4700** (2007), pp. 504–520.
URL http://www.springerlink.com/content/c82275854v4k29q0/

[2] http://wiki.portal.chalmers.se/agda/

Verification of a Message Delivery System using PRISM *

Savas Konur, Ahmed Al Zahrani and Michael Fisher

Department of Computer Science, University of Liverpool, Liverpool L69 3BX

Abstract: In this paper, we study a pervasive message delivery system, called *Scatterbox*, and formally analyze its message forwarding component.

Keywords: Pervasive systems, specification, verification,model checking

1. Introduction

Pervasive computing refers to a general class of mobile systems that can sense their physical environment and adapt their behaviour accordingly. Pervasive systems are often mobile, autonomous, distributed and concurrent, and involve humans, agents and artifacts in the system together. The success of pervasive computing depends crucially on verifying interoperability requirements for the interaction between the devices and their environment. Due to its complex nature, formal specification and verification of these requirements is very difficult and a single formal framework usually will likely be insufficient for this purpose. In the project "Verifying Interoperability Requirements in Pervasive Systems", we aim to bring together qualitative techniques, including deductive methods, model checking, and abstraction methods, with quantitative techniques, including probabilistic and performance analysis, in order to tackle the problem of verifying pervasive systems.

In this paper, we study the specification and verification of a pervasive message delivery system, called *Scatterbox* [KSC+08]. As an initial step we only consider a simplified version of the message delivery component of the Scatterbox system. While the whole system is complex and a single formal framework is insufficient, the basic delivery properties of a simplified system are suitable to be investigated using the probabilistic model checking tool, PRISM.

2. Scatterbox: A Message Delivery System

The *Scatterbox* system has been designed to serve as a test bed for context-aware computing in a pervasive computing environment. It provides a content-filtering service to its users by forwarding relevant messages to their mobile phone. The user's context is derived by tracking his/her location and monitoring his/her daily schedule. This context data is analysed, and situations are identified that indicate the user's level of interruptibility. As messages arrive, Scatterbox forwards them to subscribed users provided the user's available context suggests they are in a situation where they may be interrupted. Scatterbox's email handler connects to the user's email server as an IMAP client. Throughout the day, it downloads all unread messages and extracts information from them that will be used by Scatterbox's reasoning component to determine their importance. If Scatterbox decides to forward a message to the user's mobile device, the transmission is done through Bluetooth's Push protocol, which allows a file to be transferred between

* Work is part of an ongoing project on "Verifying Interoperability Requirements in Pervasive Systems", funded by EPSRC (EP/F033567) and involving collaboration with the universities of Birmingham and Glasgow.

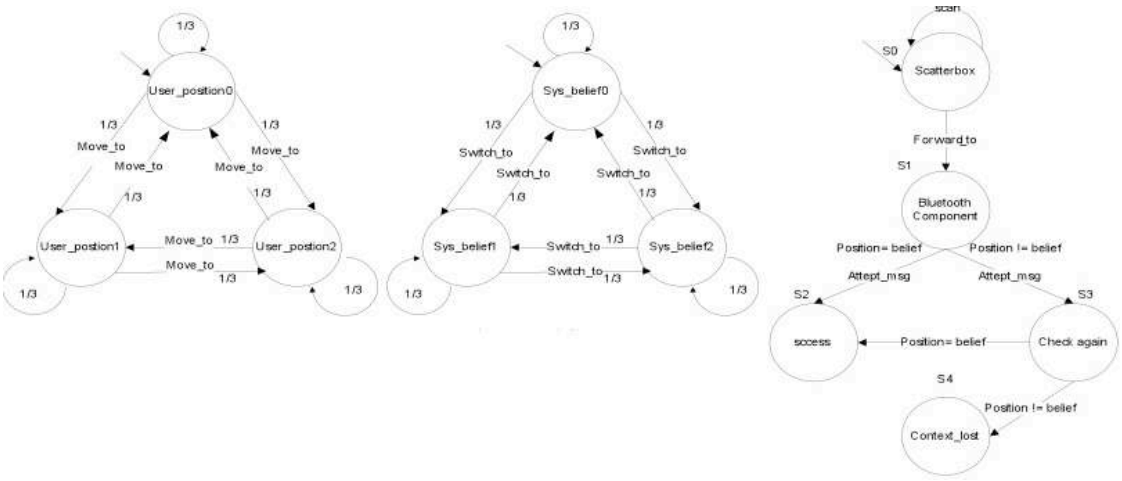


Figure 1: A simple model of the message forwarding component of the Scatterbox system.

devices. If a user's Bluetooth device is in range of a Bluetooth-enabled node, a message can be routed to that node and pushed to the mobile device. When it arrives on the user's handset, the user has the opportunity to accept or reject the message.

3. Verifying Properties of the Scatterbox System

Modeling the System. We model the message forwarding component as a transition system, in particular, as a discrete time Markov chain (DTMC). A DTMC consists of discrete states, representing the configurations of the system, and transitions governed by a discrete probability distribution on the target states.

A simplified model of the message forwarding component is shown in Figure 1. As can be seen, there is one transition system for the user movement, where the states are the actual *user position*, one transition system for the belief change, where the states are *belief* of the system on user position and one transition system for the message forwarding action. In this very simplified scenario, we assume that the one user has only three positions (*position1*, *position2* and *position3*) and that the Scatterbox system has only three belief states regarding the position of the user (*belief1*, *belief2* and *belief3*). Informally, the system forwards a message to a user position according to its belief. The belief of the system is jointly determined by the sensor information and calendar information of the user. But due to error on sensors, the system's belief of the user position, and the actual user position may be different. The transition probabilities we use are just estimated probabilities, which currently do not rely on statistical results from the real Scatterbox system.

Verifying the System using PRISM. We model the transition system of the message forwarding component in PRISM [HKNP06]. PRISM is a probabilistic model checker, which provides support for analysis of DTMCs, MDPs and CTMCs. Models are described in the PRISM modelling language, a relatively simple, state-based language, and properties are specified in the logic

PCTL [HJ94]. The system is implemented in the PRISM input language and some safety and liveness properties are specified in PCTL. As an example, consider the property denoting "what is the probability, from the initial state of the model, of a message eventually being accepted or rejected?". This property is specified in PCTL as follows: `P=?[F (s= 15 or s=16) ]` where states 15 and 16 are the "accept" and "reject" states, respectively.

4. Concluding Remarks

Our analysis has shown that PRISM can be used for the specification and verification of a simple version of the message forwarding component of the Scatterbox system. This tool might still be useful for more complex situations, such as considering user movement, transmission through SMS, etc. However, if we consider the whole system, including information access through sensors, security issues, e-mail classification etc., a single formalization tool will not be sufficient for the formal analysis of the system. Therefore, we will try to combine and synthesise different techniques to tackle the problem of verifying pervasive systems. The project aims to leverage the power of established techniques, notably:

- *model checking*, particularly work on extensions such as *parametrised model checking*, *infinite state model checking* and *probabilistic model checking*;
- the use of both *deductive* and *abstraction* techniques, allowing larger problems to be tackled; and
- a variety of different formalisms, including *logic*, *automata*, and *process calculi*, thus allowing descriptions of interaction, communication and synchronisation to be given at varying levels.

Although some of our effort will involve pushing each technique further, the majority of it will be on *combinations* of tools and techniques, i.e. bending and synthesising techniques such as [BDFF08] to make them give meaningful results in our case studies.

Bibliography

[BDFF08] R. H. Bordini, L. A. Dennis, B. Farwer, M. Fisher. Automated Verification of Multi-Agent Programs. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Pp. 69–78. 2008.

[HJ94] H. Hansson, B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing* 6:102–111, 1994.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proc. TACAS*. LNCS 3920, pp. 441–444. Springer, 2006.

[KSC+08] S. Knox, R. Shannon, L. Coyle, A. Clear, S. Dobson, A. Quigley, P. Nixon. Scatterbox: Context-Aware Message Management. *Revue d'Intelligence Artificielle* 22(5):549–568, 2008.

Reachability as deducibility, finite countermodels and verification

Alexei Lisitsa

Department of Computer Science, the University of Liverpool,
Ashton Building, Ashton Street,
Liverpool, L69 7ZF, UK

Abstract: We propose a simple but powerful approach to the verification of parameterized systems. The approach is based on modeling the reachability between parameterized states as deducibility between suitable encodings of states by formulae of first-order predicate logic. To establish a safety property, that is non-reachability of unsafe states, the finite model finder is used to find a finite countermodel, the witness for non-deducibility.

Keywords: automated verification, infinite state systems, parameterized systems, first-order logic, finite model finders, theorem provers

Reachability as deducibility

Let $\mathscr{S} = \langle S, \rightarrow \rangle$ be a transition system with the set of states S and transition relation $\rightarrow$. Consider encoding $e : s \mapsto \varphi_s$ of states of $\mathscr{S}$ by formulae of first-order predicate logic which satisfies the folowing property. The state s' is reachable from s, i.e. $s \rightarrow^* s'$ if and only if $\varphi_{s'}$ is the logical consequence of φ_s, that is $\varphi_s \models \varphi_{s'}$ and $\varphi_s \vdash \varphi_{s'}$[1]. Under such assumptions establishing reachability amounts to theorem proving, while deciding non-reachability becomes theorem disproving. In this paper we will focus on applications of these ideas to the automated verification of *safety* properties of *infinite state* and *parameterized* systems. Restriction to the safety properties, i.e. non-reachability of *unsafe* states means we will be mainly dealing with automated disproving. To disprove $\varphi_s \models \varphi_{s'}$ is is sufficient to find a countermodel for $\varphi_s \rightarrow \varphi_{s'}$, or, which is the same, the model for $\varphi_s \wedge \neg\varphi_{s'}$. In general, in first-order logic such a model may be inevitably infinite. Furthermore, the set of satisfiable first-order formulae in not recursively enumerable, so one can not hope for complete automation here. As a partial solution we propose to use automated *finite* model finders/builders [3]. Here we present preliminary results related to instantiations of these ideas to the verification of lossy channel systems [1] and to the verification of parameterized cache coherence protocols [4].

Verification of lossy channel systems

Lossy Channel Systems are essentially finite-state automata augmented witha finite amount of unbounded but lossy FIFO channels (queues). The messages sent via lossy channels may be lost in the transition (see definitions in [1]). The general form of the *safety* verification problem for lossy channel systems we address here is as follows [1].

[1] Here we assume standard definitions of semantic consequence $\models$ and deducibility $\vdash$ (in a complete deductive system) for the first-order predicate logic

Given: A lossy channel system $L = \langle S, s_0, A, C, M, \delta \rangle$ and a regular set $\Sigma \subseteq A^*$

Question: Does $Traces(L) \subseteq \Sigma$ hold?

We assume that Σ is effectively given by a deterministic finite automaton $M_{\bar{\Sigma}}$ which accepts the *complement* of Σ. Following the standard approach [1] we first reformulate equivalently the above question as a question on reachability: *Is it true that in $L \times M_{\bar{\Sigma}}$ no global state of the form $\langle\langle s,t\rangle, w\rangle$ with $t \in F$ is reachable?* Here $L \times M_{\bar{\Sigma}}$ is a lossy channel system, which is a *product* of L and $M_{\bar{\Sigma}}$ synchronized over actions from A, $\langle\langle s,t\rangle, w\rangle$ is a global state of $L \times M_{\bar{\Sigma}}$ and F is the set of accepting states of $M_{\bar{\Sigma}}$.

Verification via countermodel finding

In this subsection we show how to apply reachability as deducibility approach and finite countermodel finding for deciding the above reachability problem.

First, we define a translation of the product system $L \times M_{\bar{\Sigma}}$ into a formula of the first-order predicate logic $\Phi_{L \times M_{\bar{\Sigma}}}$. The global state γ of $L \times M_{\bar{\Sigma}}$ is encoded as a $n+2$-tuple of terms $\bar{t}_\gamma$ [2]. The intended meaning of the atomic formula $R(\bar{t}_\gamma)$ is "the global state γ is reachable", and the whole formula $\Phi_{L \times M_{\bar{\Sigma}}}$ axiomatizes the reachability in $L \times M_{\bar{\Sigma}}$:

Theorem 1 *The global state γ is reachable in $L \times M_{\bar{\Sigma}}$ if and only if $\Phi_{L \times M_{\bar{\Sigma}}} \vdash R(\bar{t}_\gamma)$.*

Let F be a set of accepting states of $M_{\bar{\Sigma}}$ and d_s be a constant denoting $s \in F$. Let then B be a first-order sentence $\vee_{s \in F} \exists y \exists \bar{x} R(y, d_s, \bar{x})$.

Theorem 2 *Either $\Phi_{L \times M_{\bar{\Sigma}}} \vdash B$, or there is a* finite *model for $\Phi_{L \times M_{\bar{\Sigma}}} \wedge \neg B$.*

Based on both theorems, parallel composition of theorem proving and finite model finding becomes a complete decision procedure for the safety problem for lossy channel systems. In practical experiments [5] with that procedure we have used a combination of the prover Prover9 and the model finder Mace4 [6], which provides with the convenient unified interface. See [5] for the report on the verification of Alternating Bit Protocol modeled as a lossy channel system.

Verification of parameterized cache coherence protocols

Another class of the systems to which verification via finite countermodel finding approach has been applied is parameterized cache coherence protocols [4], modeled by Extended Finite State Machines(EFSM) using *counting abstraction* [4]. The states of EFSM are n-dimensional non-negative integer vectors and transitions are affine transformations with affine pre-conditions. Denote by $\mathbb{Z}^+$ the set of non-negative integers. A set $S \subseteq \mathbb{Z}^{+n}$ is called *upwards closed* if it is of the form $\{(x_1, \ldots, x_n) \mid \wedge_{i=1\ldots n}(x_i \sim_i c_i)\}$ where $\sim_i$ is either $=$ or $\geq$ and $c_i \in \mathbb{Z}^+$. The general form of the parameterized verification problem we consider here is as follows.

Given: An EFSM model $M = \langle S, T \rangle$ with an upwards closed set $S \subseteq \mathbb{Z}^{+n}$ of the initial states, and the set of transtions T; and a finite family of the upwards closed sets of *unsafe* states $F_1, \ldots F_k \in \mathbb{Z}^{+n}$

[2] the first two terms are to represent the control states of L and $M_{\bar{\Sigma}}$, and the remaining n terms are to encode the content of the channels

Question: Is it true that *none* of the states in $F_1, \ldots, F_k$ is reachable from any of the initial states?

We define a translation of the EFSM model M to the first-order formula ϕ_M. The global state $\gamma \in \mathbb{Z}^{+n}$ of M is encoded by the n-tuple of terms using unary notation. The upwards closed sets of states are encoded by n-tuples of non-ground terms. For example, the tuple $(s(x), s(0), s(s(y)))$ encodes the set $\{(x,1,y) \mid x \geq 1, y \geq 2\}$. As before, the predicate R is used to specify the reachable global states. Further, the formula $\psi_{F_i}(\bar{x})$ expresses the reachability of states which belong to the upwards closed set F_i. The translation satisfies the following

Proposition 1 *If the above verification problem has a negative answer, that is there is an initial state $s \in S$ and an unsafe state $f \in \cup_{i=1,\ldots,k} F_i$ such that f is reachable from s in M then $\phi_M \vdash \vee_i \exists \bar{x}_i \psi_{F_i}(\bar{x}_i)$*

It follows that finding any model for $\phi_M \wedge \neg(\vee_i \exists \bar{x}_i \psi_{F_i}(\bar{x}_i))$ entails the positive answer for the above verification problem. Using the finite model finder Mace4 we have verified [5] all cache coherence protocols from [4].

Concluding remarks

In many aspects the proposed method, when using *finite* countermodels, is close to the methods based on constraints and symbolic reachability [4] and to the *regular model checking* [2]. One direction for further research is to give a formal comparison of the proposed approach with these and other relevant methods. One of the advantages of the countermodel finding method is its *modularity* - any correct model finder can be used within. Perhaps, the most interesting direction for research here is to test and evaluate whether using the *infinite* model finders [3], i.e. procedures which search for (finite representations of) infinite models, will increase the "verifying" power of the method.

Bibliography

[1] Parosh Aziz Abdulla, Jonsson B. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91-101, June 15, 1996.

[2] A. Bouajjani, B. Jonsson, M. Nilsson, T. Touili. Regular model checking. in *Proc. 12th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 2000.

[3] R. Caferra, A. Leitsch, N. Peltier, *Automated Model Building*, Applied Logic Series, 31, Kluwer, 2004.

[4] G. Delzanno. Constraint-based Verification of Parametrized Cache Coherence Protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[5] A. Lisitsa Verfication via countermodel finding `http://www.csc.liv.ac.uk/~alexei/countermodel/`

[6] W. McCune Prover9 and Mace4

Analysis of critical scalable real-time systems by means of Petri nets

Sadouanouan Malo and Annie Choquet-Geniet

University of Poitiers and ENSMA - Laboratory of Applied Computer Science
1 Av. Clement Ader BP 40109-86961 Futuroscope Chasseneuil-France

Abstract: The constantly increasing complexity and risks associated with real-time systems has been approached in different ways in recent years. Design and implementation of real time applications on multiprocessors has to take into account these factors. To cope with them, applications need to be more adaptive and evolutive. We present an approach for schedulability analysis of a real-time application that can accommodate reliable insertion of new tasks. This paper presents an approach based on the modelling of the application by a Petri net with terminal markings. Time is here modelled by means of the earliest firing rule. The main expected results are techniques for extraction of valid and scalable schedules within the terminal marking graph.

Keywords: real-time systems, multiprocessor scheduling, scalability, Petri nets

I Introduction

In this work we consider analysis based on Petri nets. The approach we use is an extension of the method proposed in [GC02]. Petri net based modelling is here well suited since Petri nets have been defined to model highly coupled applications. One of the main advantages of such a model is to combine interaction analysis and temporal analysis: problems linked to critical resource sharing or to communication can be addressed as well as the schedulability problem. The task set, the resources and the interactions between tasks are modelled in a classical way [CG06, Pet81]. Then, considering the earliest firing rule [Sta90] enables to introduce time. It allows the implementation of a logical clock. Finally, terminal markings enable to take deadlines into account. The terminal-marking graph built from this net represents exactly the set of valid schedules and then it is possible to extract optimal schedules according to some performance criteria.

We consider uniform multiprocessor platforms, with m processors. We assume that real-time applications are modelled by a set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ of periodic tasks which can communicate and share critical resources. Each task is characterized by four temporal parameters [LL73]: its first release time r_i, its period T_i, its worst-case computation time C_i, and its relative deadline D_i, which is the maximum delay allowed from the release of an instance of the task, to its completion.

We consider asynchronous task sets and we assume that $0 \leq r_i \leq T_i$ and $C_i \leq D_i \leq T_i, \forall \tau_i \in \tau$. For a set of n tasks τ, $T = lcm(T_i)_{i=1..n}$ denotes the hyperperiod of the application. The processor utilization factor characterizes the processor workload due to the task set. It is defined by $U = \sum_{i=1}^{i=n} \frac{C_i}{T_i}$. If $U > m$ the system isover-loaded and temporal faults cannot be avoided [But97].

Our aim is to introduce scalability criteria in the computation of valid schedules. We espe-

cially focus on task adjunction and fault tolerance by re-execution of tasks. We modelled the idle task $\tau_0 = (0, T(m-U), T, T)$ as an issue to control the processor idleness. The edges of the terminal-marking graph are labelled here by set of tasks. We use the alphabet $\sum_\tau = \{\tau_0, \tau_1, ..., \tau_n\}$ and the following label function $\sigma_\tau : T \to \sum_\tau$. $\forall T_{i,j,k} \in T, \sigma_\tau(T_{i,j,k}) = \tau_i$ and $\sigma_\tau(t) = \varepsilon$ for any other transition $t \in T$. Each schedule of the task system is a word ω defined as $\omega = \omega_1\omega_2...$ where for $t \in N$, $\omega_t = \{\tau_{i_1}...\tau_{i_m}\}$ is a set of tasks. ω_t corresponds to m tasks scheduled at instant t. The center of the terminal language of the Petri net $L(R, \sigma_\tau, \psi)$ corresponds to the set of the valid schedules of the modelled task system.

II Selecting scalable schedules and tasks assignment

II-1 Minimization of response times

In order to select schedules minimizing the response times of the tasks of a set RT_{min} ($RT_{min} \subset \{\tau_1, ..., \tau_n\}$), the graph is weighted as follows: consider two markings N_i and N_j connected by the edge $Edge(N_i, N_j)$ at the height h, and labelled by the set W_h. Each task of the set contributes to the edge weight. The contribution of $\tau_k \in W_h, k = 1..m$ denoted p_k corresponds to the response time of the tasks which complete execution at time h. It is defined by:

$$p_k = \begin{cases} 0 \text{ if the edge does not correspond to the completion of task } \tau_k \text{ or if } \tau_k \notin RT_{min} \\ \text{else} \begin{cases} T_k & \text{if } h - r_k \equiv 0\,[T_k] \\ h - \left\lfloor \frac{h-r_k}{T_k} \right\rfloor T_k - r_k & \text{otherwise} \end{cases} \end{cases}$$

The edge is then weighted by $Cost\_edge(N_i, N_j) = \sum_{\tau_k \in w_h} p_k$. If all edges are weighted, paths minimizing the cost are optimal with respect to the response times. To minimize the average response time, weights are allocated to nodes as follows, where N is a node: $Cost(N) = min_{N_i \in fils(N)} \{Cost(N_i) + Cost\_arc(N, N_i)\}$.

II-2 Fair distribution of idle time

We are interested here in schedules where idle time units are located as fairly as possible.This means that we want the idle task to be scheduled according to a fair scheme [BCPV96]. Let $U_0 = \frac{C_0}{T}$. If $C_0 > T$, then $U_0 = k_0 + u_0$ with $0 \leq u_0 < 1$. Then, in a fair manner, k_0 idle time must occur each time, and the remaining ones must be fairly distributed within each hyperperiod. The labelling function will then quantify the distance between the actual number of elapsed idle time units and the ideal number of idle time units. Here, weights are associated to nodes. If N_h is a node at depth h, associated to a marking M_h, we define its weight as : $Cost(N_h) = \mid T \times (m-U) - M_h(Activ_0) - (h \ mod \ T)(k_0 + u_0) \mid$. In this expression, h correspond to the time, $T \times (m-U) - M_h(Activ_0)$ corresponds to the number of elapsed idle time units for the pending hyperperiod, and $(h \ mod \ T)(k_0 + u_0)$ is the ideal number of elapsed idle time units for the pending hyperperiod.

II-3 Tasks assignment algorithm

We don't consider here in details the allocation problem. We assume that heuristics are used in order to carry out an efficient allocation process. One criteria of efficiency is the reduction of interprocessor migrations, which induce overhead due to context switches. We thus can use

the allocation process given by the algorithm 1, which generalized the heuristics proposed in [ADT08]. An assignment is a function A such that $A(\tau_i,t) = p_j$ means that task τ_i runs at time t on processor P_j.

Algorithm 1 **Input**: A valid schedule S_1 of length L
Output: An assignment A of tasks to processors
Let $S_1(1) = \{\tau_{i_1}, ..., \tau_{i_m}\}$
$A(\tau_{i_j}, 1) \leftarrow p_j$ // at $t = 1$, the assignment is arbitrary.
for $t = 2$ **to** L
  if $S_1(t) \cap S_1(t-1) = \oslash$ **Then**
    Let $S_1(t) = \{\tau_{i_1}, ..., \tau_{i_m}\}$; $A(\tau_{i_j}, t) \leftarrow p_j$ // Here again, the assignment is arbitrary.
  Else
    $\forall \tau_{i_j} \in S_1(t) \cap S_1(t-1), A(\tau_{i_j}, t) \leftarrow A(\tau_{i_j}, t-1)$.
    // τ_j must be assigned to the same processor as at time instant $t-1$.
    $\forall \tau_{i_j} \notin S_1(t) \cap S_1(t-1), A(\tau_{i_j}, t) \leftarrow p_k$ where k is the smallest integer less than or equal to m such that p_k is not allocated
  end if
end for

Bibliography

[ADT08] D. Aoun, A. Déplanche, Y. Trinquet. PFair scheduling improvement to reduce inter-processor migrations. In *Proceedings of RTNS 08*. Pp. 131–138. 2008.

[BCPV96] S. Baruah, N. Cohen, C. Plaxton, D. Varvel. Proportionate progress : a notion of fairness in resource allocation. *Algorithmica* 15:600–625, 1996.

[But97] G. Buttazzo. *Hard Real-Time Computing Systems,predictable scheduling,algorithms and applications*. Kluwer Academic Publishers, 1997.

[CG06] A. Choquet-Geniet. *Les réseaux de Petri - Un outil de modélisation*. Dunod, 2006.

[GC02] E. Grolleau, A. Choquet-Geniet. Off line computation of real time schedules by means of Petri nets. *Journal of Discrete Event Dynamic Systems* 12:311–333, 2002.

[LL73] C. Liu, J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM* 20(1):46–61, 1973.

[Pet81] J. Peterson. *Petri nets theory and the modelling of systems*. Prentice-Hall, 1981.

[Sta90] P. Starke. Some properties of timed Petri nets under te earliest firing rule. In *Advances in Petri nets '90*. LNCS 424. Springer Verlag, 1990.

Functional Reachability

C.-H. L. Ong and N. Tzevelekos

Oxford University Computing Laboratory

Abstract: What is *reachability* in higher-order functional programs? We formulate reachability as a decision problem in the setting of the prototypical functional language PCF, and show that even in the recursion-free fragment generated from a finite base type, several versions of the reachability problem are undecidable from order 4 onwards, and several other versions are reducible to each other. We characterise a version of the reachability problem in terms of a new class of tree automata introduced by Stirling at FoSSaCS 2009, called *Alternating Dependency Tree Automata* (ADTA). As a corollary, we prove that the ADTA non-emptiness problem is undecidable, thus resolving an open problem raised by Stirling. However, by restricting to contexts constructible from a finite set of variable names, we show that the corresponding solution set of a given instance of the reachability problem is regular. Hence the relativised reachability problem is decidable.

Keywords: Higher-Order Reachability, Control Flow Analysis, Alternating Dependency Tree Automata, PCF

In the simplest form, Reachability is the decision problem: Given a state of a state-transition system (e.g. an error state paired with a program point), is it reachable from the start state? Reachability testing has had a major impact in software model checking; it is now a standard approach to checking safety properties in industry. In the past decade, great strides have been made in model-checking reachability of first-order (recursive) procedural programs. Tools such as SLAM [1] and Blast [5] showcase the remarkable achievements of the computer-aided verification community in the engineering of scalable software model checkers. Perhaps surprisingly no reachability checker has yet been developed for *higher-order* programming languages such as ML, Ocaml, Haskell, F#. Indeed, to our knowledge, reachability of higher-order functional computation *per se* does not appear to have been studied in the literature. We initiate just such an investigation here.

Reachability of higher-order functional programs is quite different from that of first-order imperative programs. Functional programs are state-less, and it is unclear what their program points are (because the term being evaluated is being changed by substitution as the computation unfolds). Further, functional reachability is *contextual*: the flow of control within a (higher-order, open) term should be analysed in relation to all its *program contexts*; it is thus much more complex than graph reachability, which is what first-order reachability boils down to.

Consider the following decision problem in the rather purified setting of PCF [14] (a simply-typed λ-calculus with conditionals and recursion at all types), generated from a *finite* base type o of booleans.

> CONTEXTUAL REACHABILITY: Given a PCF term M of type A and a subterm N^α with occurrence α, is there a program context $C[\text{-}]$ such that $C[M]$ is a program (i.e. closed term of type o) and the evaluation of $C[M]$ causes control to flow to N^α?

Our starting point is the question: Is CONTEXTUAL REACHABILITY decidable? A precise (and equivalent) way to formulate the problem is to replace the subterm N^{α} in M by a distinguished error constant $\star$ — call the resultant term $M^{\star}$ — and ask if there is a PCF program-context $C[\text{-}]$ such that $C[M^{\star}]$ evaluates to $\star$. Here we regard the *principal term* $M^{\star}$ and the *context* $C[\text{-}]$ as elements of a larger language, PCF*, which is PCF augmented with $\star$, with with evaluation rules so extended as to propagate $\star$ to the top.

More generally, consider the following parameterised decision problem, where the (closed) principal term ranges over $\mathscr{L}_1$, the (applicative) context ranges over $\mathscr{L}_2$, both $\mathscr{L}_1$ and $\mathscr{L}_2$ are sublanguages of PCF*, and θ ranges over the base type $o := \{\mathtt{t}, \mathtt{f}, \star\}$.

> θ-REACH $[\mathscr{L}_1, \mathscr{L}_2]$: Given a closed $\mathscr{L}_1$-term $M : A_1 \to \cdots \to A_n \to o$, are there closed $\mathscr{L}_2$-terms $N_1, \cdots, N_n$ such that $M\overline{N}$ evaluates to θ ?

We can formulate the preceding reachability problem (equivalently) as $\star$-REACH $[\text{PCF}^{\star}, \text{PCF}]$.

For a sharper analysis, we consider the *finitary* (i.e. recursion-free) sublanguages, fPCF* and fPCF. Note that "divergence" is definable in PCF (e.g. $\mathbf{Y}_o(\lambda x.x)$) but not in fPCF. Thus we also consider fPCF$_{\perp}$, fPCF augmented with a divergence constant $\perp$. We obtain two results.

(i) *Undecidability*. By exploiting (the key lemma behind) Loader's proof of the undecidability of PCF observational equivalence [8], we show that CONTEXTUAL REACHABILITY, $\star$-REACH $[\text{PCF}^{\star}, \text{PCF}]$, $\mathtt{t}$-REACH $[\text{fPCF}_{\perp}, \text{fPCF}]$ and $\mathtt{t}$-REACH $[\text{fPCF}^{\star}, \text{fPCF}]$ (and several others) are all undecidable from order 4 onwards.

(ii) *Equivalence*. The problems $\star$-REACH $[\text{fPCF}^{\star}, \text{fPCF}]$ and $\perp$-REACH $[\text{fPCF}_{\perp}, \text{fPCF}]$ are polynomially reducible to each other. Whether they are decidable is open.

Motivated by the open problem, we analyse fPCF* computation automata-theoretically. Stirling [15] recently introduced a new kind of tree automata called *Alternating Dependency Tree Automata* (ADTA) which are an accepting device for trees with a binding relation called Σ-*binding trees*. He showed that the decision problem Higher-Order Matching is reducible to the ADTA non-emptiness problem, and asked if the latter is decidable. The second contribution of this paper is a characterisation of the problem v-REACH $[\text{fPCF}^{\star}, \text{fPCF}]$ (for $v \in o$) in terms of ADTA acceptance and ADTA non-emptiness problems. Thanks to the preceding undecidability result, we obtain the undecidability of ADTA non-emptiness as a corollary.

The characterisation is proved using a characterisation of fPCF* computation by *traversals* [12, 4, 3]. A traversal over the *full computation tree* (which is a souped-up syntax tree) of a term M, $\lambda^{\mathrm{f}}(M)$, is a certain sequence of nodes of the tree; unlike a path in the tree, a traversal can "jump" all over the tree.[1] Given a closed fPCF*-term M, we construct an ADTA that simulates traversals t over $\lambda^{\mathrm{f}}(M)$ by a set of paths that correspond to the P-views of prefixes of t. The states of the simulating ADTA are based on *variable profiles* [12], which are assertions about the value bound to a variable when control (in the form of a traversal) reaches it.

Our third contribution concerns a relativised reachability problem. By restricting to contexts constructible from a finite set of variable names, we show that the corresponding solution set of a given instance of $\star$-REACH $[\text{fPCF}^{\star}, \text{fPCF}]$ is recognisable by an alternating tree automaton, and hence regular. Thus the relativised problem is decidable. As a corollary, $\star$-REACH $[\text{fPCF}^{\star}, \text{fPCF}]$ is decidable at order 3.

[1] Using intuitions from game semantics, a traversal over $\lambda^{\mathrm{f}}(M)$ is a representation of an interaction sequence obtained by hereditarily *uncovering* (in the sense of Hyland and Ong [6, p. 341]) a play in the strategy-denotation of M.

Related work The aim of *Control Flow Analysis* (CFA) is to approximate the flow of control within a program phrase in the course of a computation (see e.g. Midtgaard's survey [11]). In a functional computation, control flow is determined by a sequence of function calls (possibly unknown at compile time); thus CFA amounts to approximating the values that may be substituted for bound variables during the computation. Since these values are (denoted by) pieces of syntax, CFA reduces to an algorithm that assigns *closures* (subterms of the examined term paired with substitutions for free variables) to bound variables. Reachability analysis and CFA are clearly related: for example, the former can aid the latter since unreachable parts of the term can be safely excluded from the range of closure assignment. There are however important differences: on one hand, CFA algorithms are *approximation algorithms* designed to address a more general problem; on the other, because CFA considers terms in isolation of their possible (program) contexts, the corresponding notion of reachability essentially amounts to reachability in the reduction graph.

Also relevant are (mainly type-theoretic [2, 7]) methods to detect *useless code*. A subterm of a term is *useless* if it does not contribute to the evaluation. State-of-the-art algorithms employ only static information, without predicting the values of bound variables or analysing control flow. Consequently, these algorithms offer even coarser approximations than CFA. Based on the fully abstract game semantics, traversals are a (particularly accurate) model of the flow of control within a term; they can therefore be viewed as a CFA method. In fact, Hankin and Malacaria [10, 9] proposed a game-semantical approach to CFA. Their work utilised a kind of traversals over *flowcharts*, a construction similar (but not identical) to Blum-Ong [4, 12].

Further details Please refer to the full conference version [13].

Bibliography

[1] T. Ball and S. K. Rajamani. The SLAM Project: Debugging system software via static analysis. In *Proc. POPL*, 2002.

[2] S. Berardi, M. Coppo, F. Damiani, and P. Giannini. Type-based useless-code elimination for functional programs. In *Proc. SAIG*, 2000.

[3] W. Blum. *The Safe Lambda Calculus*. PhD thesis, University of Oxford, Dec 2008.

[4] W. Blum and C.-H. L. Ong. Local computation of β-reduction: a concrete presentation of game semantics. 2009. Preprint, downloadable at william.famille-blum.org/research/.

[5] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Temporal-safety proofs for systems code. In *Proc. SPIN Workshop*, 2003.

[6] J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I,II,III. *Information and Computation*, 163:285–408, 2000.

[7] N. Kobayashi. Type-based useless-variable elimination. *Higher Order Symbolic Computation*, 14(2-3):221–260, 2001.

[8] R. Loader. Finitary PCF is not decidable. *Theor. Comp. Science*, 266:342–364, 2001.

[9] P. Malacaria and C. Hankin. Generalised flowcharts and games. In *Proc. ICALP*, 1998.

[10] P. Malacaria and C. Hankin. A new approach to control flow analysis. In *Proc. Compiler Construction*, 1998.

[11] J. Midtgaard. Control-flow analysis of functional programs. Tech. Report BRICS RS-07-18, DAIMI, University of Aarhus, 2007.

[12] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Proc. LICS*, 2006. Long version downloadable at users.comlab.ox.ac.uk/luke.ong/.

[13] C.-H. L. Ong and N. Tzevelekos. Functional Reachability. In *Proc. LICS*, 2009.

[14] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[15] C. Stirling. Dependency tree automata. In *Proc. FoSSaCS*, 2009.

Policy Refinement Checking (Extended Abstract)

Nikolaos Papanikolaou, Sadie Creese and Michael Goldsmith

e-Security Group, International Digital Laboratory, University of Warwick

Abstract: We introduce refinement checking for privacy policies expressed in P3P and XACML. Our method involves a translation of privacy policies to a set of process specifications in CSP, which describe how the privacy policy is enforced. The technique is described through an example involving medical data collected by a biobank.

Keywords: policy refinement, model checking, CSP, FDR, formal methods

1 Introduction

We are interested in the use of privacy policies as a means to protecting digital assets. It can be necessary to compare two related policies, particularly when these policies are expressed using different languages. However, even when two policies are expressed in the same language, it is possible that they will differ syntactically but supposedly express the same intention. If mistakes are made in the comparison of privacy policies this could result in the wrong policy being implemented and exposure to risk.

The Privacy Preferences Platform Project (P3P) at the W3C is focussed on developing machine readable XML for expressing websites' privacy policies and users' privacy preferences; this is intended to enable the use of privacy-aware browsers and to allow websites to collect and process information they may require in a fashion that respects user privacy. The policy languages developed within P3P [CDE+06] so far are lacking a formal semantics and hence are prone to inconsistency and ambiguity [Hog02]. XACML is a language for expressing role-based access control, and has been extended with a profile for expressing privacy policies, but also lacks a formal semantics. The lack of a widely accepted semantics for privacy policies is the main source of difficulty in policy comparison.

2 Our Approach

Policy refinement [MS93] is a term used to refer to the process of synthesising lower level policies from policies expressing higher level concerns. In the study of policy refinement one sees policies which achieve the same overall goal, but show different levels of abstraction in how the goal is achieved. The checking of refinements for concurrent processes, however, has been studied extensively, and there is a clear opportunity to unify the two notions of refinement, since policies are readily expressible as sets of processes that implement policy rules.

Of interest is how any two related policies can be compared. Such comparisons become necessary in practical applications, especially when the policies in question are expressed using different languages. Even when two policies are expressed in the same language, it is possible that they will differ syntactically but supposedly express the same intention. The lack of a

formal and hence unambiguous, and commonly accepted semantics for privacy policies is the main source of difficulty in policy comparison. What we are trying to address is not whether two given policies have one and the same meaning; we wish to determine whether one policy *refines* another, and we use the well–established technique of process refinement, commonly used for model checking CSP [RHB97, Gol05]. Our focus on refinement as opposed to equality of policies is justified by the need to check that one policy is a valid implementation of another, as might arise in the setting of a supply chain, for example.

The key step is to model the intention of a privacy policy using a set of interconnected CSP processes which can be generated automatically from a P3P or XACML policy. We are developing a tool to perform this translation. A top-level CSP process expresses a policy as a whole, and links together a sequence of smaller processes, each of which checks the conditions contained in policy rules. We are working on the development of a tool that converts P3P policies to CSP models and interacts with the FDR model checker to enable refinement checking for these. Extending this to accept OASIS XACML is a direction for future work. We believe this is a fruitful avenue of investigation with important practical applications.

Related work includes the definition of 'policy relations' by May et al. [MGL09], who proposed a high-level semantic framework for comparing policy outcomes. Their approach is mathematically elegant as it avoids comparing specific actions permitted by a policy, and focuses only on the outcomes of applying it. Yu et al. [YLA04] have proposed a relational semantics for P3P, in an effort to give unambiguous meaning to syntactically different expressions of a single policy. Their ideas are complementary to the approach we propose here.

3 Case Study: Verifying Privacy Policies for a Biobank

A biobank is a database in which is stored a vast amount of medical information (and even genetic material) for a large number of individuals; such a database is intended for research purposes, namely for the prevention of disease in future generations[1]. Participation in a biobank is entirely voluntary, and individuals are expected to disclose personally identifiable information. We assume that individuals disclose such information online.

Suppose that a fictitious biobank requires the following data from each participant: name, date of birth, address, ethnic origin, history of family diseases, and height. These data are subject to the following privacy policy: **(a)** the name, date of birth, address, and history of family diseases will be used by a research group working from within the biobank for internal research; **(b)** the name, ethnic origin, and height will be disclosed by the biobank to a third party, an independent agency working on a sociological experiment; and **(c)** the name, date of birth and height will be retained by the biobank for a maximum period of 30 days.

How can the biobank compare this policy (which is readily expressible in P3P or XACML) with a different policy, recommended by their legal team, which requires that all data be outsourced to the same third party, given that they are able to carry out all the research needed by the biobank? Through policy refinement checking we hope to be able to show formally the relationship between the levels of privacy provided to individuals by these two policies.

[1] See for example the UK Biobank: http://www.ukbiobank.ac.uk

Bibliography

[CDE+06] L. Cranor, B. Dobbs, S. Egelman, G. Hogben, J. Humphrey, M. Langheinrich, M. Marchiori, M. Presler-Marshall, J. M. Reagle, M. Schunter, D. A. Stampley, R. Wenning. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification. World Wide Web Consortium, Note NOTE-P3P11-20061113, November 2006.

[Gol05] M. Goldsmith. FDR2 User's Manual version 2.82. June 2005. http://www.fsel.com/documentation/fdr2/fdr2manual.pdf

[Hog02] G. Hogben. A technical analysis of problems with P3P v1.0 and possible solutions. In *Proceedings of W3C Workshop on the Future of P3P*. November 2002. Available at http://www.w3.org/2002/p3p-ws/pp/jrc.html.

[MGL09] M. J. May, C. A. Gunter, I. Lee. Strong and Weak Policy Relations. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY'09)*. London, UK, July 2009.

[MS93] J. D. Moffett, M. S. Sloman. Policy Hierarchies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communications* 11:1404–1414, 1993.

[RHB97] A. W. Roscoe, C. A. R. Hoare, R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. http://portal.acm.org/citation.cfm?id=550448

[YLA04] T. Yu, N. Li, A. I. Antón. A Formal Semantics for P3P. In *Proceedings of ACM Workshop on Secure Web Services*. Fairfax VA, USA, October 29 2004.

A Semantic Embedding of Promela-Lite in PVS

Shamim H. Ripon[1], Alice Miller[1] and Alastair F. Donaldson[2]

[1]Department of Computing Science, University of Glasgow, UK
[2]Computing Laboratory, University of Oxford, UK

Abstract: We present a case study in applying mechanical verification via theorem proving to Promela-Lite. We show how the syntax and semantics of Promela-Lite can be embedded in the general purpose theorem prover PVS, so that consistency of the syntax and semantics can be interactively proved.

Keywords: Promela-Lite, Semantics, PVS, Theorem Proving, Symmetry.

1 Introduction

Promela-Lite [DM08] is a specification language that captures the core features of Promela - the input language for the SPIN model checker [Hol03]. Unlike Promela, Promela-Lite has a rigorously defined semantics, making it a suitable vehicle for proving correctness of verification and state-space reduction techniques for Promela. The language was designed for the purpose of proving correct an automatic symmetry detection technique for Promela [DM08], which derives state-space preserving automorphisms from the text of a Promela specification, to be exploited during search to reduce the space and time requirements of model checking via *symmetry reduction* [CEF+96, ES96, ID96]. This symmetry detection technique, based on *static channel diagram analysis*, has been implemented as part of TopSPIN, a symmetry reduction package for the SPIN model checker [DM06].

The soundness of model checking depends critically on the correctness of underlying algorithms and reduction techniques. For example, an erroneous symmetry detection method may compute state-space permutations which are not structure-preserving, potentially resulting in incorrect verification results. For this reason, it is highly desirable that correctness proofs for model checking techniques, such as the proof by hand presented in [DM08], are automatically verified.

Mechanical verification is widely used as a tool to verify the syntax and the semantic models of a language. Verification of language properties may identify flaws in the language, which can be remedied to give increased confidence in the language definition. Theorem provers are heavily used as a tool to mechanically verify language properties. PVS [ORS92] is an automated framework for specification and verification. It supports high-order logic, allows abstract datatypes to model process terms, and has a strong support for induction mechanisms. PVS also supports interactive proof checking, where the user applies proof commands to simplify the goal to be proved until it can be proved automatically by its decision procedure.

In this paper, inspired by other successful attempts to embed specification languages into PVS [POS04, RB08], we show how the Promela-Lite syntax and semantics can be embedded into PVS, and how this embedding is used to interactively prove both consistency of the syntax/semantics definitions, and language properties. In particular, we concentrate on proving

theorems related to automatic symmetry detection which have previously been proved only by hand.

2 Embedding Promela-Lite in PVS

Promela-Lite is embedded into PVS in a step by step manner starting from the definition of types, syntax and semantics, and extending to the definition of various porperties and theorems of the language.

The encoding of a language in PVS requires one to define the available types of the language. We first define the primitive types `int` and `pid`, where `int` represents the integer values and `pid` denotes the process id values. Both types are defined as a *subtype* of natural numbers. A *channel* type in Promela-Lite can accept any other types including a channel type itself as its parameter, which results in a recursive type for channels. The `DATATYPE` mechanism of PVS is used to define Promela-Lite types as a datatype in PVS.

Proofs about languages which have a BNF style syntax definition often require induction over the terms of the language. PVS provides a powerful mechanism to define abstract datatypes and it generates an induction scheme for them. The BNF formed Promela-Lite syntax is defined in PVS using an abstract datatype. When these definitions are type checked, PVS generates new files containing a large number of useful definitions and properties of the datatypes, as well as an induction scheme.

Typing rules for Promela-Lite are defined to ensure that the language terms are well-formed. These are also important while proving language properties. The Promela-Lite semantics are defined in PVS in such a way to ensure that the definitions preserve the typing rules. The definition of the semantics also requires the notion of a *state* of a specification. A state can be represented as an ordered tuple consisting of current values of each variable and channel, or as a set of propositions. In PVS we represent a state as a set of variables, and values, with a mapping from variables to values which allows us to identify the current value of any variable at a state. An important part of our semantic definition is the update rules which include the rules for assignments and for reading from and writing to channels. By repeatedly applying the update rules we can define a sequence of updates.

3 Proving Language Properties

After defining the syntax and semantics of Promela-Lite, we prove some theorems and supporting lemmas which have been previously proved by hand. The *static channel diagram* associated with a Promela-Lite specification $\mathscr{P}$ is a coloured, bipartite digraph $SCD(\mathscr{P}) = (V, E, \Upsilon)$. The group of all automorphisms of $SCD(\mathscr{P})$ is denoted by $Aut(SCD(\mathscr{P}))$. We can find this group easily, using the computational group theory package, GAP [gap06].

Our ultimate goal is to prove that a subgroup G of $Aut(SCD(\mathscr{P}))$, known as the valid automorphisms of $\mathscr{P}$, is an automorphism group of $\mathscr{M}$, the underlying Kripke-structure of $\mathscr{P}$. The proof of this relationship depends on various underlying definitions and supporting lemmas. One of the main hurdles that needs to be overcome to automate the proof steps is to identify all the supporting rules and definitions that are required. We do this by extracting definitions and

lemmas from the published hand proofs. The major challenges are to define:

- permutations for expressions, guards and update statements. Separate permutation rules are needed for expressions based on the types of returned values when the expressions are evaluated. Inductive definitions are needed for guards.
- permutations of Promela-Lite channels (recursive definition).
- supporting lemmas for expressions, guards and update statements. The proof of each lemma extensively uses various definitions and permutation rules. The challenge here is to apply the rules and definitions appropriately in the proof steps. Most of these rules are not defined separately in the hand proof. Identifying these definitions requires a deep understanding of the semantics and the proof steps. We have defined permutations for expressions and guards, and completed proofs of the corresponding lemmas. We are currently proving these lemmas for update statements.

Bibliography

[CEF+96] E. Clarke, R. Enders, T. Filkorn, , S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design* 9(1–2):77–104, 1996.

[DM06] A. F. Donaldson, A. Miller. A Computational Group Theoretic Symmetry Reduction Package for the Spin Model Checker. In *AMAST'06*. Lecture Notes in Computer Science 4019, pp. 374–380. Springer, 2006.

[DM08] A. F. Donaldson, A. Miller. Automatic Symmetry Detection for Promela. *Journal of Automated Reasoning* 41:251–293, 2008.

[ES96] E. Emerson, A. Sistla. Symmetry and model checking. *Formal Methods in System Design* 9(1–2):105–131, 1996.

[gap06] GAP– Groups Algorithms and Programming, Version 4.4. Aachen, St. Andrews, http://www.gap-system.org/, 2006.

[Hol03] G. J. Holzman. *The SPIN MODEL CHECKER: Primer and Reference Manual*. Addison-Wesley, 2003.

[ID96] C. Ip, D. Dill. Better verification through symmetry. *Formal Methods in System Design* 9:41–75, 1996.

[ORS92] S. Owre, J. Rushby, N. Shankar. PVS: A Prototype Verification System. In Kapur (ed.), *CADE'92*. LNAI 607, pp. 748–752. Springer-Verlag, June 1992.

[POS04] C. L. Pombo, S. Owre, N. Shankar. A Semantic Embedding of the Ag Dynamic Logic in PVS. Technical report SRI-CSL-02-04, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 2004.

[RB08] S. H. Ripon, M. J. Butler. PVS Embedding of cCSP Semantic Models and their Relationship. In Calder and Miller (eds.), *AVoCS'08*. Pp. 128–142. 2008.

Program Verification via Extraction from Coinductive Proofs

Monika Seisenberger

Swansea University, Wales, UK

Abstract: Program extraction from formal proofs is a powerful technique for obtaining certified programs. We discuss and demonstrate the advantages of this method and show how it is supported by the interactive proof assistant MINLOG. Specifically, we focus on program extraction from coinductive proofs and applications to exact real arithmetic.

Keywords: Program Extraction, Realisability, Coinduction, Exact Real Number Computation

Motivation and Aim. Our starting point is the formal verification of algorithms for exact real numbers with respect to signed digit representation by Berger and Tie [BH09]. Their work is based on Plume's formalisation of arithmetical operations on real numbers represented as infinite streams [Plu98] and uses coinductive reasoning for the verification process. Other work on exact real number computation and its verification can be found e.g. in [CD06, EH02, GNSW07, ME07]. All these approaches have in common that the algorithms are first specified and implemented, and then, post-hum, verified. Hence, the underlying system has to formalise both, the abstract real numbers and their representations, as well as the algorithms computing these representations.

In our work we aim at the verification of algorithms like the ones mentioned above, however instead of applying the traditional specify-implement-verify method, we want to obtain the algorithms together with a correctness certificate via extraction from formal proofs. That is, we only have to reason about abstract real numbers; their representation and the corresponding algorithms are obtained automatically via the extraction process. In this way, we are able to extract new algorithms which, by construction, are provably correct. Moreover, we aim to demonstrate that this requires less formalisation effort compared to the usual approach.

Methodology and Tool support. The formalisation and program extraction process, as well as the case studies will be carried out the interactive proof assistant MINLOG [BBS+98, Sch06]. MINLOG (`www.minlog-system.de`) is based on first order natural deduction, extended by inductive data types and inductive predicates, and allows for program extraction via Kreisel's modified realisability [Kre59] from both constructive and classical proofs. Program extraction for formal systems including coinduction has been studied e.g. in [Tat98, MP05]. We present an alternative treatment of realisability for coinductive definitions which has the advantage that it allows for nesting of inductive and coinductive definitions and extracts realisers that directly correspond to programs in a lazy functional programming language (see also [BS09]). The extension of MINLOG's extraction mechanism with respect to coinductive definitions is ongoing work together with Munich University.

Bibliography

[BBS+98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, W. Zuber. Proof theory at work: Program development in the Minlog system. In Bibel and Schmitt (eds.), *Automated Deduction – A Basis for Applications*. Applied Logic Series II, pp. 41–71. Kluwer, Dordrecht, 1998.

[BH09] U. Berger, T. Hou. Coinduction for Exact Real Number Computation. In *Computability in Europe 2006*. Springer, 2009. To appear.

[BS09] U. Berger, M. Seisenberger. Program extraction via typed realisability for induction and coinduction. 2009. Submitted.

[CD06] A. Ciaffaglione, P. Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science* 351:39–51, 2006.

[EH02] A. Edalat, R. Heckmann. Computing with Real Numbers: I. The LFT Approach to Real Number Computation; II. A Domain Framework for Computational Geometry. In Barthe et al. (eds.), *Applied Semantics - Lecture Notes from the International Summer School, Caminha, Portugal*. Pp. 193–267. Springer, 2002.

[GNSW07] H. Geuvers, M. Niqui, B. Spitters, F. Wiedijk. Constructive analysis, types and exact real numbers. *Mathematical Structures in Computer Science* 17(1):3–36, 2007.

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics*, pp. 101–128, 1959.

[ME07] J. R. Marcial-Romero, M. H. Escardo. Semantics of a sequential language for exact real-number computation. *Theoretical Computer Science* 379(1-2):120–141, 2007.

[MP05] F. Miranda-Perea. Realizability for Monotone Clausular (Co)Inductive Definitions. *Electronic Notes in Theoretical Computer Science* 123:179–193, 2005.

[Plu98] D. Plume. A Calculator for Exact Real Number Computation. University of Edinburgh, 1998.

[Sch06] H. Schwichtenberg. Minlog. In Wiedijk (ed.), *The Seventeen Provers of the World*. Lecture Notes in Artificial Intelligence 3600, pp. 151–157. 2006.

[Tat98] M. Tatsuta. Realizability of Monotone Coinductive Definitions and Its Application to Program Synthesis. In Parikh (ed.), *Mathematics of Program Construction*. Lecture Notes in Mathematics 1422, pp. 338–364. Springer, 1998.