NASA Monographs in Systems and Software Engineering

Mike G. Hinchey
Jonathan P. Bowen
Ernst-Rüdiger Olderog *Editors*

Provably Correct Systems

Springer

NASA Monographs in Systems and Software Engineering

Series editor

Mike G. Hinchey, Limerick, Ireland

Mike G. Hinchey · Jonathan P. Bowen
Ernst-Rüdiger Olderog
Editors

Provably Correct Systems

Springer

Editors

Mike G. Hinchey
Lero–The Irish Software Research Centre
University of Limerick
Limerick
Ireland

Ernst-Rüdiger Olderog
Department für Informatik
Universität Oldenburg
Oldenburg
Germany

Jonathan P. Bowen
School of Engineering
London South Bank University
London
UK

ISSN 1860-0131 ISSN 2197-6597 (electronic)
NASA Monographs in Systems and Software Engineering
ISBN 978-3-319-48627-7 ISBN 978-3-319-48628-4 (eBook)
DOI 10.1007/978-3-319-48628-4

Library of Congress Control Number: 2016959748

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

The ProCoS Project (1989–1991) was funded by the European Community as a Basic Research Project, with a continuation (ProCoS II) also funded from 1992 to 1995. It was included in the ESPRIT programme of internationally collaborative research in Information Technology. The inspiration of the project was the recent completion of the Stack verification project, undertaken by Computational Logic, Inc. This was a start-up company founded and directed by Bob Boyer and J Moore, professors at the University of Texas at Austin. Both the European and the US projects sought to advance the technology of software verification by accepting the challenge of verification of components of a free-standing computer system. These included its operating system, its assembler, and its automatic verification aids, and even the hardware of a processor chip.

Many technological breakthroughs were triggered by these two challenge projects. The international collaboration which was forged by the ProCoS Project has continued under support of the individual national funding agencies. It has inspired and accelerated the automation of program verification. The resulting tools have found application in many advanced modern industries, including industrial giants in aerospace, electronics, silicon fabrication, automobiles, communications, advertising, social networks, retail sales, as well as suppliers of compilers and of operating systems and general software.

Many of the collaborators in the original ProCoS project, together with their students and followers, have contributed to this broadening of the application of the original basic research. A representative selection of their recent work was presented at the ProCoS Workshop in March 2015; and I welcome the publication of the proceedings in book form. I offer its authors and readers my best wishes for further progress in the understanding of the basic science, coupled with its broadest possible application.

Cambridge, UK
June 2016

Tony Hoare

Preface

ProCoS is the acronym for "Provably Correct Systems", a basic research project funded in two phases by the European Commission from 1989 to 1995. This project was planned by Tony Hoare (Oxford University), Dines Bjørner (DTU, Technical University of Denmark, Lyngby), and Hans Langmaack (University of Kiel). Its goal was to develop a mathematical basis for the development of embedded, real-time computer systems.

The survey paper on ProCoS presented at the conference FTRTFT (Formal Techniques in Real-Time and Fault-Tolerant Systems) 1994 states in its introduction:

> An embedded computer system is part of a total system that is a physical process, a plant, characterized by a state that changes over time. The role of the computer is to monitor this state through sensors and change the state through actuators. The computer is simply a convenient device that can be instructed to manipulate a mathematical model of the physical system and state. Correctness means that the program and the hardware faithfully implement the control formulas of the mathematical model of the total system, and nothing else. However, the opportunities offered by the development of computer technology have resulted in large, complex programs which are hard to relate to the objective of system control.

The ProCoS project developed a particular approach to mastering the complexity of such systems. Its emphasis was on proving system correctness across different abstraction layers. The inspiration for ProCoS stems from a sabbatical of Tony Hoare at the University of Austin at Texas in 1986. There he was impressed by the work of Robert S. Boyer and J Strother Moore on automated verification with their "Boyer-Moore" prover ACL2 at their company "Computational Logic, Inc." (CLI), in particular its application to a case study known as the "CLInc Stack". Discussing later with Dines Bjørner and Hans Langmaack, a project on the foundation of verification of many-layered systems was conceived: ProCoS. The different levels of abstraction studied in this project became known as the "ProCoS Tower". They comprised (informal) expectations, (formal) requirements, (formal) system specifications, programs (in the "occam" programming language), machine code (for the "transputer" microprocessor), and circuit diagrams (described using "netlists").

During the final deliverable for the first phase of ProCoS, Tony Hoare wrote in 1993:

> In summary, our overall goal is not to produce a single verified system or any particular verified language or compiler, but rather to advance the state of the art of systematic design of complex heterogeneous systems, including both hardware and software; and to concentrate attention on reducing the risk of error in the specification, design and implementation of embedded safety critical systems.

In the first phase, the ProCoS project comprised seven partners: Oxford University, Technical University of Denmark at Lyngby, Christian-Albrechts Universität Kiel, Universität Oldenburg, Royal Holloway and Bedford New College, Århus University, and the University of Manchester. In the second phase (ProCoS II), the team consisted of the first four original partners. The EU funding of ProCoS was relatively small. During the second phase only one researcher at each of the four partner sites was funded, but many more students and researchers at these sites contributed to the goals.

ProCoS was much influenced by the work of two Chinese scientists contributing to the project at Lyngby and Oxford: Zhou Chaochen and He Jifeng.

Zhou Chaochen and Anders P. Ravn initiated a major conceptual development of ProCoS: the Duration Calculus, an interval-based logic for specifying real-time requirements. The first paper on it was published by Zhou Chaochen, Tony Hoare and Anders P. Ravn in 1991. The types of durational properties that can be expressed in the Duration Calculus were motivated by the case study of a gas burner that was defined by E.V. Sørensen from DTU in collaboration with a Danish gas burner manufacturer. He Jifeng cooperated closely with Tony Hoare on a predicative approach to programming that led to the book "Unifying Theories of Programming" (UTP) published in 1998. The work on UTP has attracted a number of researchers and led to a series of symposiums on this topic.

To bridge the gap from requirements to programs, a combination of specification techniques for data and processes with transformation rules was developed by the group of E.-R. Olderog in Oldenburg. The topic of correct compilers, exemplified for the translation of an occam-like programming language to transputer machine code, was investigated in the group of Hans Langmaack in Kiel. Oxford contributed an algebraic approach to compiling verification.

Associated with the ProCoS project was an EU-funded ProCoS Working Group (1994–1997) of 25 academic and industrial partners interested in provably correct systems, arranging various meetings around Europe.

Other associated national projects in the United Kingdom included the "safemos" project (1989–1993), a UK EPSRC project on "Provably Correct Hardware/Software Co-design" (1993–1996), and an EPSRC Visiting Fellowship on "Provably Correct Real Time Systems" (1996–1997). Associated travel funding to encourage collaboration included ESPRIT/NSF ProCoS-US initiative on "Provably Correct Hardware Compilation" with Cornell University in the US, and KIT (Keep in Touch) grants with UNU/IIST in Macau (1993–1998) and PROCORSYS with the Federal University of Pernambuco in Brazil (1994–1997).

Impact

An extension of the Duration Calculus to cover continuous dynamical systems was led by Anders P. Ravn and Hans Rischel at DTU to contribute to the initial research on hybrid systems.

From 1992 until 1997, Dines Bjørner was the founding director of UNU-IIST, the International Institute for Software Technology of the United Nations University in Macau. Ideas from the ProCoS project flourished at the institute and were taken up by researchers from Asia working there. Also, a number of scientists associated with ProCoS visited UNU-IIST or had research posts for several years, including He Jifeng and Zhou Chaochen. From 1997 until 2002, Zhou Chaochen succeeded Dines Bjørner as the director of UNU-IIST, during the time of transition of Macau from a Portuguese to a Chinese city. Regrettably, in 2013 the United Nations decided to disband academic staff at UNU-IIST.

A number of young ProCoS contributors pursued academic careers. Martin Fränzle and Markus Müller-Olm, students in Kiel during the ProCoS project, are now professors at the universities of Oldenburg and Münster, respectively. Also, Debora Weber-Wulff and Bettina Buth, at Kiel during the ProCoS project, are now professors in Berlin and Hamburg, respectively. Michael Schenke, a ProCoS contributor at Oldenburg, is now a professor in Merseburg. Augusto Sampaio, during ProCoS working on his Ph.D. at Oxford on an algebraic approach to compilation during ProCoS, has became a professor at the University of Pernambuco, Brazil. Paritosh K. Pandya, working at Oxford during the ProCoS project, has become a professor at the Tata Institute of Fundamental Research in Mumbai, India. Zhiming Liu, who during ProCoS times spent a year as a postdoc at DTU and later was a researcher at UNI-IIST, is now professor at the Southwest University in Chongqing, China.

The collaborative project Verifix (*Construction and Architecture of Verifying Compilers*) directed by Gerhard Goos, Friedrich von Henke and Hans Langmaack and funded 1995–2004 by the German Research Foundation (DFG) deepened research on compiler correctness begun in the ProCoS project.

The large-scale Transregional Collaborative Research Center AVACS (*Automatic Verification and Analysis of Complex Systems*), directed by Werner Damm and funded by German Research Foundation (DFG) during the period 2004–2015, continued research pioneered in ProCoS but with emphasis on automation and for wider classes of systems. The collaborating sites were Oldenburg, Freiburg, and Saarbrücken. AVACS comprised of nine projects in the areas of real-time systems, hybrid systems, and systems of systems.

A series of conferences called VSTTE (*Verified Systems—Theories, Tools and Experiments*), was initiated by a vision for a Grand Challenge project formulated by Tony Hoare and Jay Misra in July 2005.

The ProCoS project and its related initiatives have inspired a number of books, including the following:

- He Jifeng, *Provably Correct Systems—Modelling of Communicating Languages and Design of Optimized Compilers*, McGraw-Hill, 1994.

- Jonathan P. Bowen (ed.), *Towards Verified Systems*, Elsevier Science, Real-Time Safety Critical Systems Series, 1994.
- Mike G. Hinchey and Jonathan P. Bowen (eds.), *Applications of Formal Methods*, Prentice Hall, Series in Computer Science, 1995.
- C.A.R. Hoare and He Jifeng, *Unifying Theories of Programming*, Prentice Hall, Series in Computer Science, 1998.
- Jonathan P. Bowen and Mike G. Hinchey, *High-Integrity System Specification and Design*, Springer, FACIT Series, 1999.
- Zhou Chaochen and Michael R. Hansen, *Duration Calculus—A Formal Approach to Real-Time Systems*, Springer, 2004.
- E.-R. Olderog and Henning Dierks, *Real-Time Systems—Formal Specification and Automatic Verification*, Cambridge University Press, 2008.

Structure of this Book

In September 2013, Jonathan Bowen and Ernst-Rüdiger Olderog met at the Festschrift Symposium for He Jifeng in Shanghai and discussed the possibility of having a workshop celebrating 25 years of ProCoS. This idea materialized with the help of Mike Hinchey in March 2015, when a two-day ProCoS Workshop with around 40 invited researchers and 25 presentations on the topic of "Provably Correct Systems" took part in the rooms of the BCS in London. This book consists of 13 chapters mainly describing recent advances on "Provably Correct Systems", based on presentations at that workshop. Each paper has been carefully reviewed by three to five reviewers. The chapters address the following topics:

- Historic Account,
- Hybrid Systems,
- Correctness of Concurrent Algorithms,
- Interfaces and Linking,
- Automatic Verification,
- Run-time Assertions Checking,
- Formal and Semi-formal Methods, and
- Web-Supported Communities in Science.

Historic Account

In the note "ProCoS: How it all Began—as seen from Denmark", Dines Bjørner opens his diary and shows entries by Tony Hoare during a meeting of IFIP Working Group 2.3 at Château du Pont d'Oye in Belgium in 1987. The author explains that this was a first draft on the content of ProCoS.

Hybrid Systems

Martin Fränzle, Yang Gao, and Sebastian Gerwinn review in Chap. "Constraint-Solving Techniques for the Analysis of Stochastic Hybrid Systems" definitions of (parametric) stochastic hybrid automata as needed for reliability evaluation. The authors then discuss automatic verification and synthesis methods based on arithmetic constraint solving. The chapters are able to solve step-bounded stochastic reachability problems and multi-objective parameter synthesis problems, respectively.

Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao, and Liang Zou introduce in Chap. "MARS: A Toolchain for Modelling, Analysis and Verification of Hybrid Systems" the toolchain MARS for Modelling, Analysing and verifying hybrid Systems. Using MARS, they build executable models of hybrid systems using the industrial standard environment Simulink/Stateflow, which facilitates analysis by simulation. The toolchain includes a translation of Simulink/Stateflow models to Hybrid CSP and verification using an interactive prover for Hybrid Hoare Logic.

Correctness of Concurrent Algorithms

John Derrick, Graeme Smith, Lindsay Groves, and Brijesh Dongol study in Chap. "A Proof Method for Linearizability on TSO Architectures" the correctness of non-atomic concurrent algorithms on a weak memory model, the TSO (Total Store Order) model. They show how linearizability is defined on TSO, and how one can adapt a simulation-based proof method for use on TSO. Their central result is a proof method that simplifies simulation-based proofs of linearizability on TSO.

Interfaces and Linking

E.-R. Olderog, A.P. Ravn, and R. Wisniewski investigate in Chap. "Linking Discrete and Continuous Models, Applied to Traffic Manoeuvrers" the interplay between discrete and continuous dynamical models, and combine them with linking predicates. The topic of linking system specifications at different levels of abstraction was central to the ProCoS project. However, here the application area is more advanced: traffic manoeuvrers of multiple vehicles on highways.

Zhiming Liu and Xin Chen discuss in Chap. "Towards Interface-Driven Design of Evolving Component-Based Architectures" how software design for complex evolving systems can be supported by an extension of the rCOS method for refinement of component and object systems. It shows the need for a suitable interface theory and of multi-modelling notations for the description of multi-viewpoints of designs. This requires a theoretical foundation in the style of Unifying Theories of Programming as proposed by Tony Hoare and He Jifeng.

Automatic Verification

J Strother Moore presents in Chap. "Computing Verified Machine Address Bounds During Symbolic Exploration of Code" an abstract interpreter for machine address expressions that attempts to produce a bounded natural number interval guaranteed to contain the value of the expression. The interpreter has been proved correct by the ACL2 theorem prover. The author discusses the interpreter, what has been proved about it by ACL2, and how it is used in symbolic reasoning about machine code.

Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann describe in Chap. "Engineering a Formal, Executable x86 ISA Simulator for Software Verification" a formal, executable model of the x86 instruction-set architecture (ISA). They use this model to reason about x86 machine-code programs. Validation of the x86 ISA model is done by co-simulating it regularly against a physical x86 machine.

Jens Otten and Wolfgang Bibel present in Chap. "Advances in Connection-Based Automated Theorem Proving" calculi to automate theorem proving in classical and some important non-classical logics, namely first-order intuitionistic and first-order modal logics. These calculi are based on the connection method. The authors present details of the leanCoP theorem prover, a very compact PROLOG implementation of the connection calculus for classical logics. leanCoP has also been adapted to non-classical logics by integrating a prefix unification algorithm.

Run-Time Assertion Checking

Frank S. de Boer and Stijn de Gouw extend in Chap. "Run-Time Deadlock Detection" run-time assertions by attribute grammars for specifying properties of message sequences. These assertions are used in a method for detecting deadlocks at run-time in both multi-threaded Java programs and systems of concurrent objects.

Tim Todman and Wayne Luk present in Chap. "In-Circuit Assertions and Exceptions for Reconfigurable Hardware Design" a high-level approach to adding assertions and exceptions in a hardware design targeting FPGAs (Field Programmable Gate Arrays). They allow for imprecise assertions and exceptions to trade performance for accurate location of errors.

Formal and Semi-formal Methods

Bettina Buth reports in Chap. "From ProCoS to Space and Mental Models – A Survey of Combining Formal and Semi-Formal Methods" on work influenced by the ProCoS project. Systems from the application areas of space and aerospace are analysed using suitable abstractions to CSP specifications and the FDR model checker.

Web-Supported Communities in Science

Jonathan P. Bowen studies in Chap. "Provably Correct Systems: Community, Connections, and Citations" the building and support of scientific communities and collaboration, especially online, visualized graphically and formalized using the Z notation, including the concept of a "Community of Practice". His examples are drawn from the ProCoS project.

In summary, we hope that you enjoy this volume, providing a selection of research developments and perspectives since the original ProCoS initiatives of the 1990s. Further ProCoS-related information can be found online under:

http://formalmethods.wikia.com/wiki/ProCoS

Limerick, Ireland — Mike G. Hinchey
London, UK — Jonathan P. Bowen
Oldenburg, Germany — Ernst-Rüdiger Olderog

Acknowledgements

The following reviewed papers in these proceedings:
Wolfgang Bibel, Darmstadt University of Technology, Germany
Simon Bliudze, EPFL, Switzerland
Jonathan P. Bowen, London South Bank University, UK
Bettina Buth, HAW Hamburg, Germany
Michael Butler, University of Southampton, UK
Ana Cavalcanti, University of York, UK
Frank de Boer, CWI, The Netherlands
Willem-Paul de Roever, University of Kiel, Germany
John Derrick, Unversity of Sheffield, UK
Martin Fränzle, University of Olderburg, Germany
Anthony Hall, Independent consultant, UK
Jifeng He, East China Normal University, China
Warren Hunt, University of Texas, USA
Cliff Jones, Newcastle University, UK
Zhiming Liu, Southwest University, China
Annabelle McIver, Macquarie University, Australia
Dominique Méry, University of Lorraine, LORIA, France
Ernst-Rüdiger Olderog, University of Olderburg, Germany
Jan Peleska, University of Bremen, Germany
Anders P. Ravn, Aalborg University, Denmark
Augusto Sampaio, Federal university of Pernambuco, Brazil
Elizabeth Scott, University of London, UK
Jianqi Shi, National University of Singapore, Singapore
Marina Waldén, Åbo Akademi University, Finland
Jim Woodcock, University of York, UK
Naijun Zhan, Institute of Software, Chinese Academy of Sciences, China
Huibiao Zhu, East China Normal University, China

Contents

Part I
Historic Account

ProCoS: How It All Began – as Seen from Denmark

Dines Bjørner

Abstract I reminisce over an episode at the 9–13 November 1987 IFIP WG2.3 meeting at Château du Pont d'Oye in Belgium — and at what followed.

I had given a half hour presentation of how we, in Denmark, had developed a compiler for the full Ada programming language. My presentation had evolved around a single slide showing boxes and arrows between these, all properly labeled. Edsger W. Dijkstra had railed during my short presentation against my using diagrams — despite my claiming that boxes denoted certain kinds of algebras and arrows certain kind of morphisms between these. After my talk there was a coffee break. Tony Hoare took me aside. Asked permission to write in my note book. And this is what he wrote:

As he wrote it, Tony carefully explained what he was after. To me that became the day of conception of the first ProCoS project. I leave it to you to decipher the characteristic handwriting of Tony. OCC is some programming language. So is CSP. A specification maps programs in OCC into programs in CSP. VLSI is a language for specifying VLSI designs. Its semantics, in terms of CSP, is behav.

Now find, i.e., develop, a VLSI implemented machine, M, and some OCC code which maps into traces of the behaviour of M such that for all programs, D, in OCC, the VLSI machine implements the OCC specification correctly.

At the end of writing and narrating this, Tony asked: *should we try get an ESPRIT BRA project around this idea ?* We did.

• • •

To me a deciding moment of the project occurred during our Bornholm workshop. Prof. E.V. Sørensen had given a talk in which he sketched, from the background of his field, Circuit Theory, some ideas about handling digital signal transitions. I

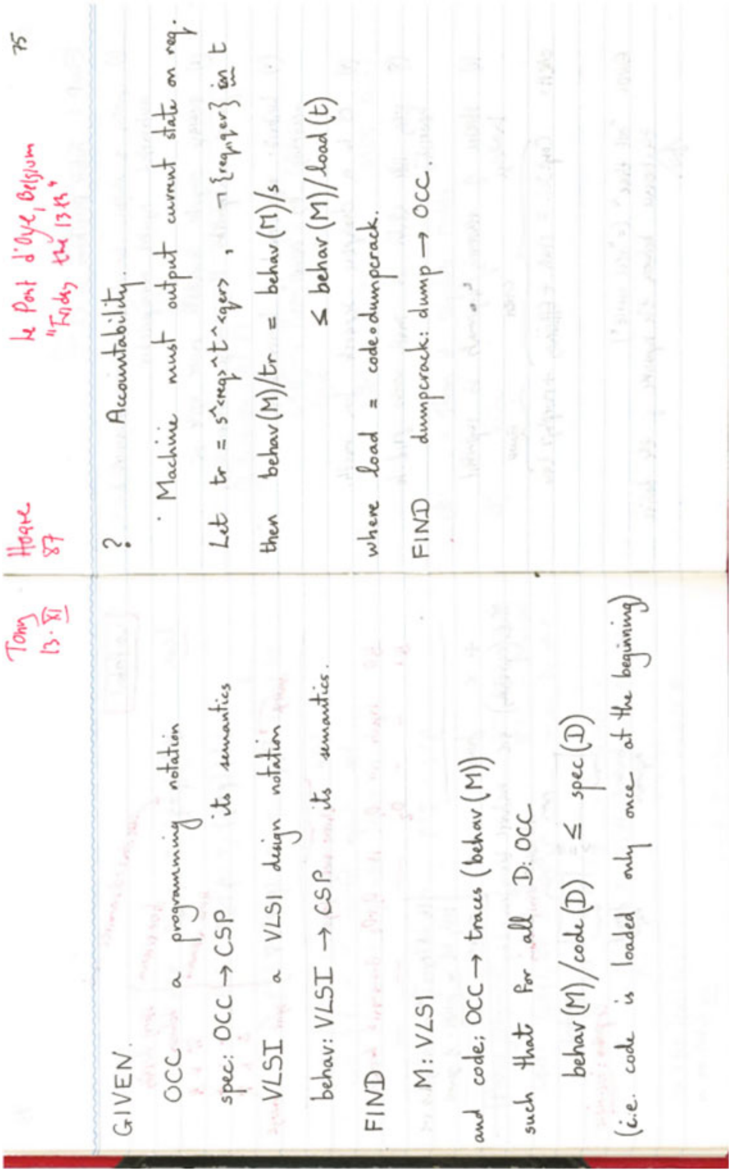
Paper read at the *BCS-FACS ProCoS Workshop on Provably Correct Systems*, London, UK, 9–10 March 2015.
The original version of this chapter was revised: For detailed information please see Erratum. The erratum to this chapter is available at 10.1007/978-3-319-48628-4_14

D. Bjørner (✉)
Fredsvej 11, 2840 Holte, Denmark
e-mail: mike.hinchey@lero.ie

believe that Erling's talk gave impetus to the Duration Calculus. During the break, after EVS' talk, I saw Anders (Ravn), Tony and Zhou discussing, it appeared, the evolving DC ideas.

● ● ●

During the ProCoS "25'th Anniversary" event, in London, 9–10 March, 2015, in the afternoon, after my morning presentation of the above "reminiscences", Tony wrote in the same notebook:



Tony
13 · XI

Hoare
87

le Pont d'Oye, Belgium
"Friday the 13th"

75

GIVEN.

OCC a programming notation

spec: OCC → CSP its semantics

VLSI a VLSI design notation

behav: VLSI → CSP its semantics.

FIND

M: VLSI

and code: OCC → traces (behav (M))

such that for all D: OCC

behav (M) / code (D) ≤ spec (D)

(i.e. code is loaded only once at the beginning)

? Accountability.

· Machine must output current state on req.

Let tr = s^<req>^t^<qer> , ¬ {req, qer} in t

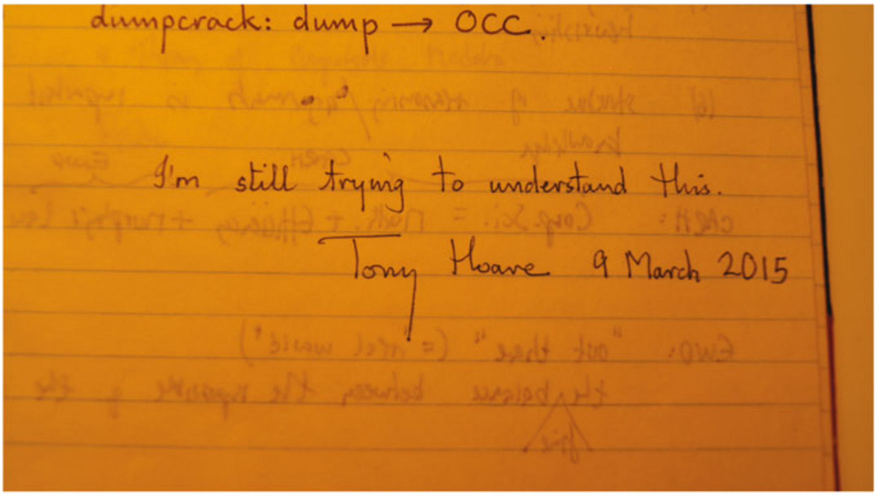
then behav (M) / tr = behav (M) / s

≤ behav (M) / load (t)

where load = code∘dumpcrack.

FIND dumpcrack: dump → OCC.

I leave this to you to work on !



dumpcrack: dump → OCC.

I'm still trying to understand this.

Tony Hoare 9 March 2015

Part II
Hybrid Systems

Constraint-Solving Techniques for the Analysis of Stochastic Hybrid Systems

Martin Fränzle, Yang Gao and Sebastian Gerwinn

Abstract The ProCoS project has been seminal in widening the perspective on verification of computer-based systems to a coverage of the detailed interaction and feedback dynamics between the embedded system and its environment. We have since then seen a steady increase both in expressiveness of the "hybrid" modeling paradigms adopting such an integrated perspective and in the power of automatic reasoning techniques addressing relevant fragments of logic and arithmetic. In this chapter we review definitions of stochastic hybrid automata and of parametric stochastic hybrid automata, both of which unify the hybrid view on system dynamics with stochastic modeling as pertinent to reliability evaluation, and we elaborate on automatic verification and synthesis methods based on arithmetic constraint solving. The procedures are able to solve step-bounded stochastic reachability problems and multi-objective parameter synthesis problems, respectively.

1 Introduction

An increasing number of the technical artifacts shaping our ambience are relying on often invisible embedded computer systems, rendering embedded computers the most common form of computing devices today. The vast majority — 98% according

This research has partially been funded by the German Research Foundation through the Collaborative Research Action SFB-TR 14 "Automatic Verification and Analysis of Complex Systems" (AVACS, www.avacs.org) and the Research Training Group DFG-GRK 1765: "System Correctness under Adverse Conditions" (SCARE, `scare.uni-oldenburg.de`).

M. Fränzle (✉) · Y. Gao
Department of Computing Science, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
e-mail: fraenzle@informatik.uni-oldenburg.de

Y. Gao
e-mail: yang.gao@informatik.uni-oldenburg.de

S. Gerwinn
OFFIS Institute for Information Technology, Oldenburg, Germany
e-mail: sebastian.gerwinn@offis.de

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_2

to www.artemis-ju.eu — of all processing elements manufactured goes to embedded applications, where they monitor and control all kinds of physical processes. Such interactions of the virtual with the physical world range from traditional control applications, like controlling an automotive powertrain, over computer-controlled active safety systems, like the anti-locking brake, the electronic stability program, or recently pedestrian detection integrated with emergency braking capabilities, to the vision of cyber-physical networks bringing even remote physical processes into our sphere of control.

Even to the general public, it has become evident that such immersion of computing elements into physical environments renders their functionality critical in many respects: critical to the function of the overall product, where a malfunction or undesired interaction of the embedded system may render the whole product partially or totally dysfunctional; critical to the performance of the overall product, where embedded control may influence power consumption, environmental impact, and many more performance dimensions; finally safety-critical, as causal chains mediated by the physical environment may propagate software faults and thus endanger life and property. A direct consequence is that correctness (in a broad sense) of embedded systems most naturally is defined in terms of the possible physical impact of its interaction with the environment. This insight marked a paradigm shift in the attitude to software correctness at the times when the ProCoS project was conceived a good quarter of a century ago. Rather than saying that correct software ought to infallibly implement some abstract algorithm or establish correctness properties in terms of conditions on intrinsic program variables (and other notions intrinsic to the algorithm, like termination), correctness became defined in terms of observables of external physical processes underlying an independent dynamics extrinsic to and only partially controllable by the software.

Within the ProCoS project, this issue was taken up by Zhou Chaochen and Anders Ravn, who first together with Tony Hoare defined the Duration Calculus [9] and later together with Hans Rischel and Michael R. Hansen extended it to cover hybrid discrete-continuous phenomena [10, 34]. While the former took the perspective of durational metric-time properties at the interface between embedded system and environment, thus not really covering environmental dynamics, the extension to Extended Duration Calculus in [10] permitted integration about environmental variables and thus formulation of integral equations, as equivalent to initial-value problems of ordinary differential equations. It thus is an example of a formal model permitting to model and analyze the tight interaction, and hence feedback dynamics, of the discrete switching behavior of embedded systems and the continuous dynamics of the physical environment as well as of continuous control components embedded into it. A related model also confining the description of environmental dynamics to ordinary differential equations and also adopting a qualitative view of the embedded system as a potentially demonically non-deterministic discrete computational device is Hybrid Automata [1]. Such models support *qualitative* behavioral verification in the sense of showing that a system behaving according to its nominal dynamics would *never* be able to engage in an undesired behavior, which, however, is a goal unlikely to be achieved in practice. First, designing systems to that level of correctness may be prohibitively expensive or it may be impractical due to necessary inclusion of components lacking tangible models of nominal behavior, like, e.g., computer vision and image classification algorithms. Second, even when possible in principle, the systems verified to that level will necessarily eventually deviate from their nominal behavior due to, e.g., component failures. Reflecting the consequential need for *quantitative* verification, stochastic variants of hybrid-system models have been suggested, like Probabilistic Hybrid Automata (PHA) [35] or Stochastic Hybrid Automata (SHA) [25]. In such extensions, either discrete actions can feature probabilistic branching or continuous evolution can evolve stochastic along, e.g., stochastic differential equations, or both.

The resulting models are inherently hard to analyze, as they combine various sources of undecidability, like state reachability in even the simplest classes of hybrid automata and the fragments of arithmetic induced by ordinary differential equations, with the necessity of reasoning about probability distributions over complexly shaped and sometimes not even first-order definable carriers, like the reachable states. It is thus obvious that exact automatic analysis methods for properties of interest, like the probability of reaching undesirable states, are impossible to attain. Nevertheless, safe approximations can be computed effectively, and often prove to be of sufficient accuracy to answer relevant questions with scrutiny, like certifying that the probability of reaching undesirable states remains below a given quantitative safety target. The pertinent techniques do either rely on state-space discretization by safe abstraction, e.g. Hahn et al.'s approach [43], or on constraint solving for stochastic logic involving arithmetic [15, 19], or on massive simulation paired with statistical hypothesis testing, beginning with Younes' seminal work [42]. We will in the remainder of this chapter report on our contributions to the constraint-based approach, thereby building on a series of results obtained over the past decade.

Structure of the Chapter

In the next section, we provide an introduction to a class of stochastic hybrid automata featuring stochasticity —paired with non-determinism or parametricity— in their transitions. In Sect. 3, we move on to the depth-bounded safety analysis of such stochastic hybrid automata. The underlying technology is an extension of satisfiability-modulo-theory solving (SMT, [4]) to Stochastic Satisfiability-Modulo-Theory (SSMT) akin to the extension of Propositional Satisfiability (SAT) to Stochastic Propositional Satisfiability (SSAT) suggested by Papadimitriou and Majercik [28, 33]. Section 4, finally, turns to the problem of multi-objective parameter synthesis in parametric variants of stochastic hybrid automata, which we solve by a machine-learning style integration of simulation and arithmetic constraint solving.

2 Stochastic Hybrid Transition Systems

The model of hybrid automata [1, 2, 22, 26] has been suggested as a formal set-up for analyzing the the interaction of discrete and continuous processes in hybrid-state dynamical systems. They combine pertinent formalism for describing discrete,

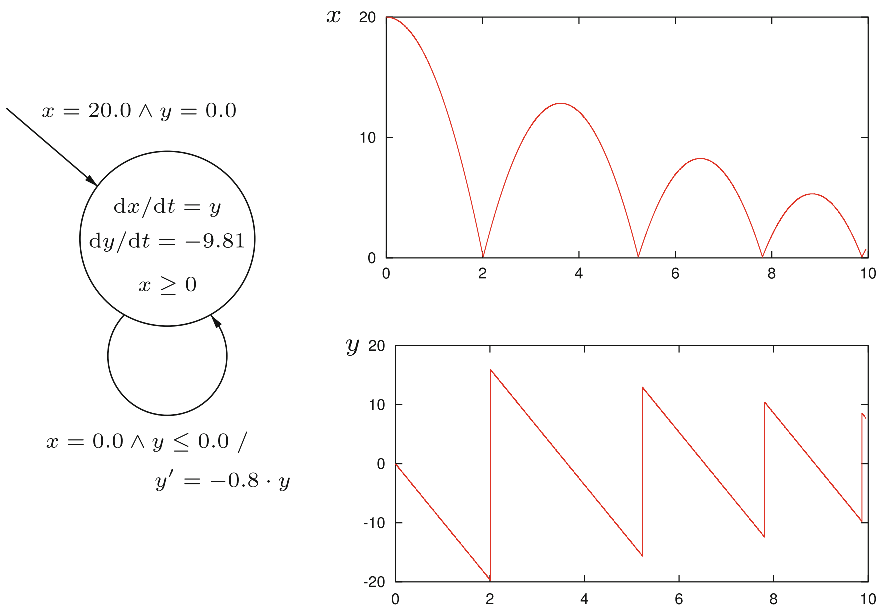
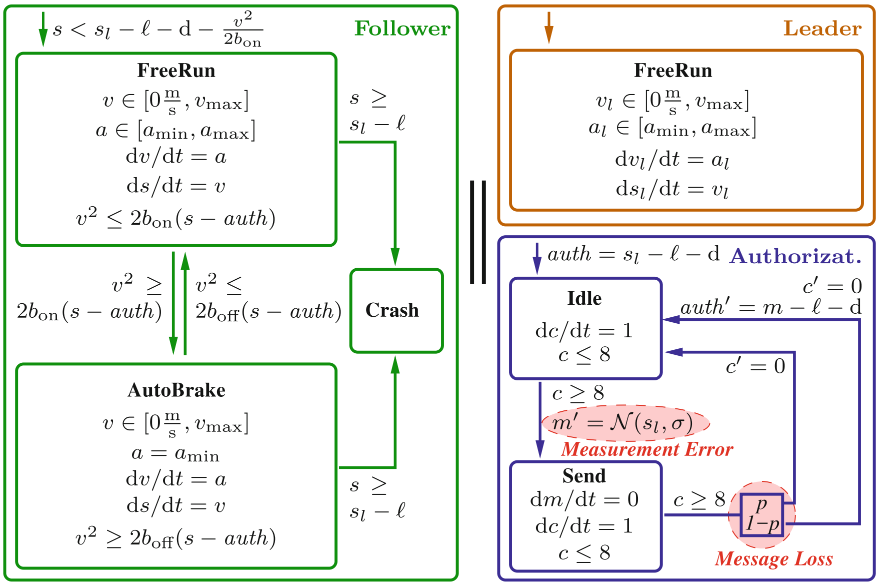

$x = 20.0 \wedge y = 0.0$
$\mathrm{d}x/\mathrm{d}t = y$
$\mathrm{d}y/\mathrm{d}t = -9.81$
$x \geq 0$
$x = 0.0 \wedge y \leq 0.0$ /
$y' = -0.8 \cdot y$
x
y

Fig. 1 A simple hybrid automaton (*left*) and a trajectory thereof (*right*)

computational and continuous, mostly physical or control-oriented dynamical processes by extending finite automata with a vector of continuous variables and "decorating" them with ordinary differential equations in each location and assignments to these extra variables upon transitions. A simple hybrid automaton and its associated dynamic behavior, which is a piece-wise continuous trajectory, are depicted in Fig. 1.

While this model permits the analysis of deterministic as well as uncertain hybrid-state systems, as the latter can be modeled by various forms of non-determinism in the automata, like uncontrolled inputs, non-deterministic transition selection, or parameter ranges in the differential (in-)equations, it is confined to *qualitative* behavioral verification in the sense of showing that a system behaving according to its nominal dynamics would never be able to engage in an undesired behavior. This ideal, however, is hardly achieved in practice, as systems strictly adhering to their nominal behavior would either be prohibitively expensive or even infeasible to design. Qualitative verification consequently is indicative of the nominal behavior only, yet does not cover the full set of expected behaviors of the system under design, which includes (traditionally rare, but with the advent of trained classifiers in, e.g., computer vision systems for automated driving increasingly frequent) deviations from nominal behavior also.

As the expected low to moderate frequency of deviations from nominal behavior would not justify the same demonic view of uncertainties in system dynamics as adopted in qualitative verification, namely that every single abnormal behavior that


Follower
$s < s_l - \ell - \mathrm{d} - \frac{v^2}{2b_{\mathrm{on}}}$
FreeRun
$v \in [0\frac{\mathrm{m}}{\mathrm{s}}, v_{\max}]$
$a \in [a_{\min}, a_{\max}]$
$\mathrm{d}v/\mathrm{d}t = a$
$\mathrm{d}s/\mathrm{d}t = v$
$v^2 \leq 2b_{\mathrm{on}}(s - auth)$
$s \geq s_l - \ell$
$v^2 \geq 2b_{\mathrm{on}}(s - auth)$
$v^2 \leq 2b_{\mathrm{off}}(s - auth)$
Crash
AutoBrake
$v \in [0\frac{\mathrm{m}}{\mathrm{s}}, v_{\max}]$
$a = a_{\min}$
$\mathrm{d}v/\mathrm{d}t = a$
$\mathrm{d}s/\mathrm{d}t = v$
$v^2 \geq 2b_{\mathrm{off}}(s - auth)$
$s \geq s_l - \ell$
Leader
FreeRun
$v_l \in [0\frac{\mathrm{m}}{\mathrm{s}}, v_{\max}]$
$a_l \in [a_{\min}, a_{\max}]$
$\mathrm{d}v_l/\mathrm{d}t = a_l$
$\mathrm{d}s_l/\mathrm{d}t = v_l$
Authorizat.
$auth = s_l - \ell - \mathrm{d}$
Idle
$\mathrm{d}c/\mathrm{d}t = 1$
$c \leq 8$
$c' = 0$
$auth' = m - \ell - \mathrm{d}$
$c' = 0$
$c \geq 8$
$m' = \mathcal{N}(s_l, \sigma)$
Measurement Error
Send
$\mathrm{d}m/\mathrm{d}t = 0$
$\mathrm{d}c/\mathrm{d}t = 1$
$c \leq 8$
$c \geq 8$
p
$1-p$
Message Loss

$$v_{\max} = 83.4\tfrac{\mathrm{m}}{\mathrm{s}}, \quad \ell = 200\mathrm{m}, \quad \mathrm{d} = 400\mathrm{m},$$
$$a_{\min} = -1.4\tfrac{\mathrm{m}}{\mathrm{s}^2}, \quad a_{\max} = 0.7\tfrac{\mathrm{m}}{\mathrm{s}^2}, \quad b_{\mathrm{on}} = -0.7\tfrac{\mathrm{m}}{\mathrm{s}^2}, \quad b_{\mathrm{off}} = -0.3\tfrac{\mathrm{m}}{\mathrm{s}^2}$$

Fig. 2 Model of moving-block train control in ETCS level 3 including relevant random disturbances (encircled areas) in the form of measurement error in positional information and possible message loss, after [17]

might be possible would render the system design incorrect, *quantitative* counterparts to qualitative hybrid models and verification methods have been developed. The corresponding *probabilistic* or *stochastic* models provide more concise quantitative information about the uncertainties involved in terms of probabilities. To incorporate this kind of information, both the underlying models and the corresponding analysis techniques have to be adapted. Verifying reachability and safety properties within this extended setting then corresponds to obtaining statements about the probability of these properties to be satisfied.

To illustrate the challenges arising in incorporating the information about random disturbances into the model of hybrid automata, we show a model model mimicking distance control at level 3 of the European train control systems ETCS in Fig. 2. The idea of the control system is to switch to an automatic braking mode “AutoBrake” initiating a controlled emergency deceleration whenever the necessary deceleration rate for coming to a standstill at a safety distance (400 m) behind the preceding train exceeds a threshold value ($-0.7\ \frac{\mathrm{m}^2}{\mathrm{s}}$). The function of this control system, however, is impeded by measurement noise in determining train positions and the risk of message loss between trains, as positions are determined locally in a train and announced via train-to-train communication. The extended hybrid model depicted in Fig. 2 incorporates these stochastic disturbances. More specifically, the perturbed measurement of the position of the leading train is characterized by a normal distribution $\mathcal{N}(s_l, \sigma)$ centered around the true position s_l and the measurement process itself is modeled by copying this skewed image of the physical entity s_l into its real-time image m. This is in contrast to a typical nominal model, where the controller may be modeled as having direct access to the physical entities. Additionally, unreliable communication is also considered, i.e., the communication of resultant movement authorities is allowed to fail with probability 0.1. The resulting model is called a *Stochastic Hybrid Automaton (SHA)* [25, 35]. Note that the automaton in Fig. 2 features both non-determinism and stochasticity in its transitions, with the former being interpreted demonically.

For such models, we are interested in solving two problems, which will be the subjects of Sects. 3 and 4:

1. Given a stochastic hybrid automaton A and a set of undesirable states G in the state set of A, determine whether the probability of reaching G stays below a given safety target ε. Given that non-determinism is interpreted demonically, the probability of reaching G thereby has to be determined w.r.t. a most malicious adversary resolving the non-deterministic choices.
2. Given a stochastic hybrid automaton featuring parameters in its probability distributions, determine whether there is a parameter instance satisfying a design objective in terms of expectations on some cost and/or reward variables in the hybrid system.

Formally, SHA are infinite-state Markov Decision Processes (MDP), where the infinite-state behavior is induced by the hybrid discrete-continuous state dynamics, while the MDP property arises from the interplay of stochastic and non-deterministic choices. A stochastic hybrid system (in its continuous-time variant) thus interleaves

1. *continuous flows* arising while residing in a discrete location and being governed by the differential (in-)equation and the invariant condition assigned to the location with
2. *immediate transitions* featuring a guard condition on the continuous variables, a deterministic or non-deterministic state update w.r.t. both some continuous variables and possibly the discrete successor location, and potentially a series of randomized updates to continuous variables are the successor location.

As a suitable semantic basis for the automatic analysis of hybrid automata featuring stochastic behavior, we can consequently base our investigations on a more abstract definition of hybrid-state transition systems featuring stochastic behavior, a form of infinite-state Markov Decision Process (MDP). In full generality, such a (parametric) hybrid stochastic transition system comprises the following:

1. A finite set $D = \{d_1, \ldots .d_m\}$ of discrete variables. Discrete variables range over $\mathbb{Z}$.[1]

[1]The reader might expect to rather see finite sub-ranges of $\mathbb{Z}$ or other finite sets as domains. To avoid cluttering the notation, we abstained from this. It should be noted that this does not induce a loss of generality, as not all of $\mathbb{Z}$ need to be dynamically reachable.

2. A finite set $C = \{x_1, \ldots, x_n\}$ of continuous variables. Continuous variables range over $\mathbb{R}$.[2]
3. A finite (and possibly empty) set $P = \{p_1, \ldots, p_k\}$ of parameters with associated range $\theta \subseteq \mathbb{R}^k$.
4. An initial state $i \in \Sigma$, where $\Sigma = \mathbb{Z}^m \times \mathbb{R}^n$ is the state set of the transition system.
5. A finite set $T = \{t_1, \ldots, t_l\}$ of stochastic transitions. Each such transition comprises a non-deterministic guarded assignment, expressed as a pre-post relation in $\Sigma \times \Sigma$, followed by a finite (possibly empty) sequence of stochastic assignments to individual variables, which are executed in sequence and may depend on the preceding ones and on the parameters.

Traditional stochastic hybrid automata diagrams, as depicted in Figs. 2 and 6, can easily be interpreted as instances of this model by interpreting both their continuous flows and immediate transitions as stochastic transitions. To this end, please note that stochastic transitions need not contain a proper stochastic part, but may also be just non-deterministic or even deterministic.

In order to achieve a uniform treatment of discrete and continuous stochastic assignments, we equip $\mathbb{R}$ with the Lebesgue measure and $\mathbb{Z}$ with cardinality of its subsets as a measure. Given this convention, we can uniformly write $\int_a^b p(x)\mathrm{d}x$ for determining the probability mass assigned by a density (or, in the discrete case, a distribution) p to the interval $[a, b]$, as the measure is understood. Note that in the discrete case, $\int_a^b p(x)\mathrm{d}x = \sum_{x=a}^{b} p(x)$ due to the particular choice of the measure for $\mathbb{Z}$. This permits us to uniformly treat densities over the continuum and distributions over discrete carriers as densities. A density over domain X, where X is either $\mathbb{R}$ or $\mathbb{Z}$, is a measurable function $\delta : X \to \mathbb{R}_{\geq 0}$ with $\int_{-\infty}^{\infty} \delta(x)\mathrm{d}x = 1$. We denote by P_X the set of all densities over X. A stochastic assignment for a variable $v \in D \cup C$ with its associated domain $V \in \{\mathbb{Z}, \mathbb{R}\}$ is a mapping $sa_v : \theta \to \Sigma \to P_V$. It assigns to each parameter instance and each state a density of the successor values for v.

We are ultimately interested in determining the probability of reaching a certain set $G \subset \Sigma$ of goal states or the expectation of a function $f : \sigma \to \mathbb{R}$. As the former is a special case of the latter, using the characteristic function χ_G[3] as reward, it suffices to define expectations.

Given that a non-deterministic assignment simply is a relation between pre- and post-states, i.e., a subset of $\Sigma \times \Sigma$ defining both the transition guard (due to possible partiality of the relation) and the (possibly non-deterministic) update to all the variables, depth-bounded expectations in the infinite-state MDP mediated by the stochastic hybrid transition system can now be defined inductively by means of a Bellmann backward induction [5] as follows: The (best-case) expectation $\mathcal{E}_f^k(\sigma, \theta)$ of reward f over k steps of the transition system under parameterization θ and starting from state $\sigma \in \Sigma$ is

[2]As for discrete variables, this does not exclude the possibility that only a bounded sub-range may dynamically be reachable.

[3]Defined as $\chi_G(\sigma) = \begin{cases} 1 & \text{if } \sigma \in G, \\ 0 & \text{if } \sigma \notin G. \end{cases}$

$$\begin{aligned}
\mathcal{E}_f^0(\sigma,\theta) &= f(\sigma)\ ,\\
\mathcal{E}_f^{k+1}(\sigma,\theta) &= \max_{t=\langle na,sa_1,\ldots,sa_n\rangle\in T}\ \max_{\sigma_1\in\Sigma \text{ such that } (\sigma,\sigma_1)\in na}\\
&\qquad \int\ldots\int sa_1^\theta(\sigma_1)(\sigma_2)\cdots sa_n^\theta(\sigma_n,)(\sigma_{n+1})\mathcal{E}_f^k(\sigma_{n+1},\theta)\mathrm{d}\sigma_1\ldots\mathrm{d}\sigma_{n+1}\ .
\end{aligned}$$

Here, na denote the non-deterministic assignment and sa_i^θ denotes the effect of a stochastic assignment, resp., in a transition $t=\langle na, sa_1,\ldots,sa_n\rangle$. The effect sa^θ of a stochastic assignment sa to variable v is

$$sa^\theta(\sigma)(x)=\begin{cases}\sigma(x) & \text{if } x\neq v,\\ sa(\theta)(\sigma)(v) & \text{if } x=v.\end{cases}$$

Taken together, the non-deterministic transition selection as well as the non-deterministic assignment thus implements an oracle maximizing rewards, while the stochastic assignments just implement their stochastic transition kernels.

3 Bounded Reachability Checking for Stochastic Hybrid Automata

In order to analyze Stochastic Hybrid Automata (SHA) models, like the one depicted in Fig. 2, we need techniques being able to analyze their intrinsic combination of stochastic dynamics and infinite-state behavior. Formally, SHA are infinite-state Markov Decision Processes (MDP), where the infinite-state behavior is induced by the hybrid discrete-continuous state dynamics, while the MDP property arises from the interplay of stochastic and non-deterministic choices (e.g., concerning a in Fig. 2). As verification tools for finite-state MDP are readily available, a particular technique is to use abstraction for obtaining a safe finite-state overapproximation, subsequently verifying the properties of interest on the abstraction, as pursued e.g. in [17]. A more direct approach along the lines of bounded model checking (BMC) [6, 21] in its variant for hybrid automata [3, 12, 23] is to encode the stochastic behavior within the constraint formula. This requires more expressive constraint logic then the satisfiability-modulo-theory calculi used in the case of qualitative verification [3, 12, 23, among others], which have been pioneered by Fränzle, Hermanns, and Teige under the name *Stochastic Satisfiability Modulo Theory (SSMT)* [15].

3.1 Stochastic Satisfiability Modulo Theory

The idea of modeling uncertainty in satisfiability problems was first proposed within the framework of propositional satisfiability (SAT) by Papadimitriou, yielding

Stochastic SAT (SSAT) [33], a logic featuring both existential quantifiers and randomized quantifiers allowing to express $1\frac{1}{2}$ player games (one strategic, one randomized player). This work has been lifted to Satisfiability Modulo Theories (SMT) by Fränzle, Teige et al. [15, 38], providing a logic called Stochastic Satisfiability Modulo Theory (SSMT) permitting symbolical reasoning about bounded reachability problems of probabilistic hybrid automata (PHA). Instead of being true or false, an SSAT or SSMT formula Φ has a probability as semantics. This quantitative semantics reflects the probability of satisfaction of Φ under optimal resolution of the non-random quantifiers. SSAT and SSMT permit concise description of diverse problems combining reasoning under uncertainty with data dependencies. Applications range from AI planning [27, 29, 30] to analysis of PHA [15]. A major limitation of the SSMT-solving approach pioneered by Teige [37] is that all quantifiers (except for implicit innermost existential quantification of all otherwise unbound variables) are confined to range over finite domains. As this implies that the carriers of probability distributions have to be finite, a large number of phenomena cannot be expressed within that SSMT framework, such as continuous noise or measurement error in hybrid systems. To overcome this limitation, our recent work [19] relaxes the constraints on the domains of randomized variables, now also admitting quantification over continuous domains and continuous probability distributions in SSMT solving.

The approach is based on a combination of the iSAT arithmetic constraint solver [13] with branch-and-prune rules for the quantifiers generalizing those suggested in [14, 37]. Covering an undecidable fragment of real arithmetic involving addition, subtraction, multiplication and transcendental functions, measuring solution sets exactly by an algorithm obviously is infeasible. The solving procedure therefore approximates the exact satisfaction probability of the formula under investigation and terminates with a conclusive result whenever the approximation gets tight enough to answer the question whether the satisfaction probability is above or below a target value specified by the user, e.g., a safety target.

We will subsequently introduce the logic manipulated by our solver, then explain the solving procedure, and finally demonstrate its use for analyzing stochastic hybrid automata.

The syntax of the input language of the solver, which is SSMT formulae over continuous quantifier domains, CSSMT for short, agrees with the discrete version from [15], except that continuous quantifier ranges are permitted.

Definition 1 An SSMT formula with continuous domain (CSSMT formula) is of the form $\Phi = \mathcal{Q} : \varphi$, where

- $\mathcal{Q} = Q_1 x_1 \in \mathrm{dom}(x_1) \ldots Q_n x_n \in \mathrm{dom}(x_n)$ is a quantifier prefix binding a sequence $x_1, \ldots, x_n$ of quantified variables. Here $\mathrm{dom}(x_i)$ denotes the domain of variable x_i, which may be an interval over the reals or integers. Each Q_i either is an existential quantifier $\exists$ or a randomized quantifier Я_{p_i} assigning a computable probability density function p_i over $\mathrm{dom}(x_i)$ to x_i.[4]

[4]In practice, we offer a selection from a set of predefined density functions over the reals. For discrete carriers, we offer the ability to write arbitrary distributions by means of enumeration.

- φ is a quantifier-free formula over some arithmetic theory $\mathcal{T}$, in our case involving addition, subtraction, multiplication, exponentiation, and transcendental functions, as supported by the iSAT SAT-modulo-theory solver. As definitional translations [40] permit to rewrite arbitrary formulae to equisatisfiable conjunctive normal forms (CNF), we may w.l.o.g. assume that φ is in conjunctive normal form (CNF), i.e., that φ is a conjunction of clauses, and a clause is a disjunction of (atomic) arithmetic predicates of the forms $v \sim c$ or $v = e$, where v is a variable, $\sim \in \{<, \leq, =, \geq, >$ a relational symbol, c a rational constant, and e an expression over variables involving addition, subtraction, multiplication, and transcendental functions. φ is also called the *matrix*[5] of the formula.

The semantics of CSSMT formulae is defined by a $1\frac{1}{2}$-player game mediated by the alternation in the quantifier prefix: following the sequence in the quantifier prefix and respecting the possible moves permitted by the quantifier domains, an existential player tries to maximize the overall probability of satisfaction while her randomized opponent subjects the existential player's strategy to random disturbances. Formally, the semantics can be defined by a Bellman backward induction over the game graph [5], akin to the semantics for SSAT [33]:

Definition 2 The semantics of a CSSMT formula $\Phi = \mathcal{Q} : \varphi$ is defined by its probability of satisfaction $Pr(\Phi)$ as follows, where ε denotes the empty quantifier prefix:

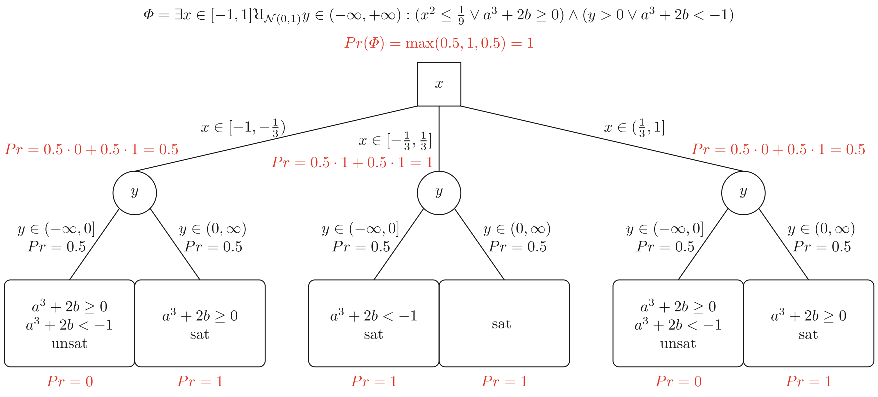
$$\begin{aligned}
Pr(\varepsilon : \varphi) &= 0, \text{ if } \varphi \text{ is unsatisfiable;} \\
Pr(\varepsilon : \varphi) &= 1, \text{ if } \varphi \text{ is satisfiable;} \\
Pr(\exists x_i \in \mathrm{dom}(x_i)\, \mathcal{Q} : \varphi) &= \sup_{v \in \mathrm{dom}(x_i)} Pr(\mathcal{Q} : \varphi[v/x_i]) \ ; \\
Pr(\text{Я}_{p_i} x_i \in \mathrm{dom}(x_i)\, \mathcal{Q} : \varphi) &= \int_{\mathrm{dom}(x_i)} Pr(\mathcal{Q} : \varphi[v/x_i]) p_i(v) \mathrm{d}v \ .
\end{aligned}$$

Here, $\mathcal{Q}$ denotes an arbitrary (possibly empty) quantifier prefix and $\varphi[v/x_i]$ signifies the substitution of value v into φ.

According to Definition 2, the semantics yields the *supremal probability of satisfaction $Pr(\Phi)$*, which is computed by resolving the quantifiers from left to right, whereby existential quantifiers are resolved by an optimal strategy guaranteeing highest reward and randomized quantifiers yield the expectation of the remaining formula. For a quantifier-free formula, and thus also after all quantifiers have been resolved, the probability of satisfaction of the matrix ϕ is associated with its satisfiability.

Example 1 Figure 3 exemplifies the semantics by depicting a simplified image of the infinitely branching tree spanned by the domains of the individual quantifiers. Equivalent branches have been collapsed, which is signified by the intervals associated to branches in the graphics. The graphics shows the game tree constructed according to the semantics of CSSMT formula $\Phi = \exists x \in [-1, 1] \text{Я}_{\mathcal{N}(\prime,\infty)} y \in (-\infty, +\infty)$:

[5]In SSAT parlance, this is the body of the formula after rewriting it to prenex form and stripping all the quantifiers.



$\Phi = \exists x \in [-1,1] \text{Я}_{\mathcal{N}(0,1)} y \in (-\infty,+\infty) : (x^2 \leq \frac{1}{9} \vee a^3+2b \geq 0) \wedge (y > 0 \vee a^3+2b < -1)$
$Pr(\Phi) = \max(0.5, 1, 0.5) = 1$
x
$x \in [-1, -\frac{1}{3})$
$Pr = 0.5 \cdot 0 + 0.5 \cdot 1 = 0.5$
$x \in [-\frac{1}{3}, \frac{1}{3}]$
$Pr = 0.5 \cdot 1 + 0.5 \cdot 1 = 1$
$x \in (\frac{1}{3}, 1]$
$Pr = 0.5 \cdot 0 + 0.5 \cdot 1 = 0.5$
y
$y \in (-\infty, 0]$ $Pr = 0.5$
$y \in (0, \infty)$ $Pr = 0.5$
$a^3+2b \geq 0$ $a^3+2b < -1$ unsat
$a^3+2b \geq 0$ sat
$a^3+2b < -1$ sat
sat
$Pr = 0$
$Pr = 1$

Fig. 3 Semantics of a CSSMT formula depicted as a quantifier tree with agglomerated branches

$(x^2 \leq \frac{1}{9} \vee a^3 + 2b \geq 0) \wedge (y > 0 \vee a^3 + 2b < -1)$, where $\mathcal{N}(0, 1)$ refers to the normal distribution with mean value 0 and variance 1. Semantically, Φ determines the maximum probability of the matrix across all values of x between $[-1, 1]$ when the values of y are distributed according to a standard normal distribution, i.e., determines an optimal choice of x as strategy for the existential player. We have grouped the uncountably infinitely many instances of the quantified variables into a finite set of branches reflecting cases not distinguished by the matrix. In the first branch, the domain of x is split into three parts, i.e., $x \in [-1, -\frac{1}{3})$, $x \in [-\frac{1}{3}, \frac{1}{3}]$ and $x \in (\frac{1}{3}, 1]$. For each part, we branch the domain of y into two parts. When all the quantified variables are resolved, we can check the satisfiability. For example, the leftmost leaf can be annotated with probability of satisfaction of 0, because $x \in [-1, -\frac{1}{3}]$ and $y \in (-\infty, 0]$ for this branch and the matrix consequently cannot be satisfied. When all the branches have been annotated, we can propagate the probability according to the corresponding quantifiers towards the root of the tree. For example, as y is distributed according to a standard normal distribution, the probability that $y \in (-\infty, 0]$ is 0.5. If we combine the probability from bottom to top and choose maximum value across the branches for x, as x is existentially quantified, then we obtain the probability of satisfaction of Φ, which in this simple case (but of course not generally) is 1. $\square$

The Bellman backward induction inherent to Definition 2 seems to suggest building tool support based on branching the quantifier tree and calling appropriate SAT-modulo-theory (SMT) solvers on the (many) instances of the matrix thus evolving, as sketched in the example. This, however, is impractical for two reasons:

1. when continuous quantifier domains are involved, the number of branches to be spawned, and thus of SMT problems to be solved, would be uncountably infinite,[6]

[6]To this end please note that collapsing equivalent branches, as pursued in Fig. 3, can only be done *after* solving the instances of the matrix and thus only is an option in cases where continuity arguments (or similar) permit generalizations from samples to neighborhoods.

2. even in the case of merely discrete domains, the number of branches is strictly exponential in the quantifier depth and thus rapidly becomes prohibitive for the bounded model-checking (BMC) problems we want to solve, which feature a quantifier depth proportional to the depth of BMC.

We do consequently need more efficient means of solving CSSMT formulae, which are subject of the next section.

3.2 CSSMT Solving

We will now expose a practical algorithm for solving a CSSMT formula. As the exact probability $Pr(\mathcal{Q} : \varphi)$ of satisfaction is not computable in case the matrix φ stems from an undecidable fragment of arithmetic, as usual in the hybrid-system domain, we formulate the goal of solving as an approximate decision problem. The problem we want our solving engine to resolve therefore is formalized as follows:

Definition 3 Given a CSSMT formula $\Phi = \mathcal{Q} : \phi$, a reference probability $\varepsilon \in [0, 1]$, and a desired accuracy $\delta \geq 0$, a procedure which upon termination returns

- "GE", if $Pr(\Phi)$ is greater than or equal to $\varepsilon + \delta$;
- "LE", if $Pr(\Phi)$ is less than or equal to $\varepsilon - \delta$;
- "GE" or "Inconclusive", if $Pr(\Phi) \in [\varepsilon, \varepsilon + \delta]$;
- "LE" or "Inconclusive", if $Pr(\Phi) \in [\varepsilon - \delta, \varepsilon]$.

is called sound. It is called quasi-complete if it terminates whenever $\delta > 0$.

A sound and quasi-complete solver for CSSMT thus is required to yield a definite (and correct) answer whenever the actual probability of satisfaction is separated from the acceptance threshold ε by at least δ. If closer to the threshold than δ, it may provide inconclusive, yet never counter-factual answers.

Our method for solving CSSMT formulae is intuitively split into three distinct —yet overlapping in practice— phases:

1. *Quantifier branching*: In this phase, each quantified variable's domain is covered by a finite interval-partitioning, thereby branching a *collapsed* quantifier tree akin to that depicted in Fig. 3. In contrast to the case depicted in Fig. 3, we do, however, neither make sure that the individual multi-dimensional cells (i.e., product of intervals) thus obtained contain points indistinguishable by the matrix in the sense of all yielding the same truth value, nor use the same partitioning in all sub-trees. Due to the latter, we are not confined to regular gridding of the n-dimensional variable space, which helps in adaptive local refinement enhancing precision.
2. *Paving*: For each multi-dimensional cell C generated in the previous phase, we generate two sets of sub-boxes[7] of the cell: one set $\underline{C}$ under-approximating the

[7] As usual in interval constraint solving, we call any product of intervals with computer-representable bounds a *box*.

set of points $p \in C$ satisfying the matrix and another set $\overline{C}$ over-approximating the set of points $p \in C$ satisfying the matrix. Such covers can be obtained by established paving techniques from interval analysis, e.g., by using the RealPaver tool [20]. The sum of the —easily computable due to box shape— measures $\mu(B)$ of the boxes $B \in \underline{C}$ (or analogously $B \in \overline{C}$) provide a lower estimate l_C (upper estimate u_C, resp.) of the probability of satisfaction over C.
The measure $\mu(B)$ of a box $B = \prod_{i=1}^{n}[a_i, b_i]$, where n coincides with the number of quantified variables $x_1, \ldots, x_n$, thereby is defined as

$$\mu(B) = \prod_{i=1}^{n} \mu_i([a_i, b_i]), \text{ where}$$

$$\mu_i(X) = \begin{cases} \int_X \nu_n(x)\mathrm{d}x & \text{if } x_n \text{ is randomly quantified with density } \nu_n\,, \\ 1 & \text{if } x_n \text{ is existentially quantified.} \end{cases}$$

Note that this measure does not expose an effect of existential quantification, as B is an axis-parallel box such that the choices of existential variables impose no constraints on the random variables.

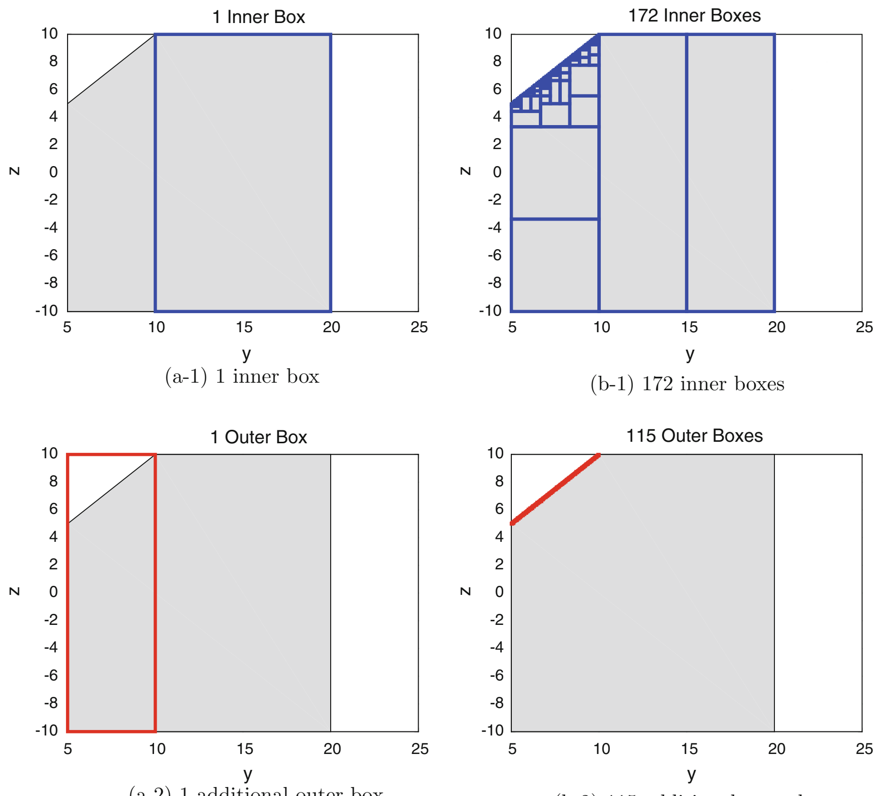
3. *Projection and lifting*: Given the bounds l_C and u_C for the satisfaction probability over each cell C, the final phase recursively synthesizes the overall satisfaction probability over the full domain by combining cells according to the quantifier prefix. If C and D are (necessarily neighboring) cells together forming a larger box-shaped cell (i.e., a product of intervals) E, then their probability masses can be combined as follows: if l_C, l_D are the respective lower and u_C, u_D the respective upper estimates of the satisfaction probabilities, and if C and D are adjacent in direction of variable x, then

$$l_E = \begin{cases} \max\{l_C, l_D\} & \text{if } x \text{ is existentially quantified,} \\ l_C + l_D & \text{if } x \text{ is randomly quantified,} \end{cases}$$

$$u_E = \begin{cases} \max\{u_C, u_D\} & \text{if } x \text{ is existentially quantified,} \\ u_C + u_D & \text{if } x \text{ is randomly quantified.} \end{cases}$$

In practice, these three phases should, however, be interleaved in order to provide adaptive refinement of covers in quantifier branching. That is, new cells are generated —and thus, the quantifier tree from phase 1 expanded— if and only if the computation of basic set measures in phase 2 or the computation of combined measures in phase 3 yield overly large differences between lower (l_C) and upper (u_C) probability estimates for some cell. In that case, the cell will be split into two in order to facilitate a sharper approximation of the actual satisfaction probability within the cell. Vice versa, local estimates generated in phase 3 can be used for pruning expansion of the quantifier tree from phase 1, generalizing Teige's effective search-space reduction [37] to the CSSMT case. To this end, it should be noted that residual cells need not be investigated if their total probability mass does not suffice to lift a partially computed (due to phase 2) mass above the acceptance threshold ε, or if the threshold already is exceed even without their contribution, or if alternative branches can be decided to yield better reward, to name just a few of numerous pruning rules rendering the procedure computationally tractable. The interested reader may refer to [37] for details of such pruning rules.

After generating an upper estimate u and a lower estimate l for the whole domain, all that remains is to check for their relation to the threshold ε: as u is a safe upper approximation of $P(\phi)$, we can report "LE", i.e., $P(\phi) \leq \varepsilon$ whenever $u \leq \varepsilon$. Similarly, as l is a safe lower approximation of $P(\phi)$, we can report "GE", i.e., $P(\phi) \geq \varepsilon$ whenever $l \geq \varepsilon$. If, however, $l < \varepsilon < u$ then the test is inconclusive, which we are



(a-1) 1 inner box

(b-1) 172 inner boxes

(a-2) 1 additional outer box

(b-2) 115 additional outer boxes

Fig. 4 Two-dimensional projections of the inner and outer approximations for the solution set of matrix $x > 3 \wedge y \leq 20 \wedge x^2 > 49 \wedge y \geq z$ over computation cell $x \in [7, 10]$, $y \in [5, 25]$ and $z \in [-10, 10]$ (corresponding to outermost frame). The *gray area* marks the exact set of models of the matrix within the cell. The *left side* gives a very rough approximation of the solution set by one inner box (a-1) and two outer boxes (a-1 plus a-2) for the over- and under-approximation, respectively. The *right side* provides a much tighter inner (b-1) and outer (b-1 plus b-2) approximation by 172 and 287 boxes, respectively

allowed to report iff $\varepsilon - l < \delta$ and $u - \varepsilon < \delta$. If the latter is not the case then we have to refine the cover of the quantifier domains by computation cells, which we obviously do by splitting those cells having the highest difference between their upper and lower probability estimates, i.e., the ones featuring the lowest accuracy.

Example 2 We consider the paving phase for the CSSMT formula

$$\begin{aligned}\Phi = \exists\, x \in [-10, 10]\, \rotatebox[origin=c]{180}{R} y \in \mathcal{U}[5, 25]\, \rotatebox[origin=c]{180}{R} z \in \mathcal{U}[-10, 10] :\\ (x > 3 \vee y < 1) \wedge (z > x^2 + 2 \vee y \le 20) \wedge\\ (x^2 > 49 \vee y > 7x) \wedge (x < 6 \vee y \ge z)\ ,\end{aligned}$$

where $\mathcal{U}[a, b]$ refers to uniform distribution with carrier $[a, b]$. If the paving procedure generates just one box for the under- and two for the over-approximation, as shown in Fig. 4(a-1 and a-2), the measures returned are $l = 0.5$ and $u = 0.75$. A more precise estimation will be obtained by generating more boxes in the paving: the 172 inner boxes and 287 outer boxes of Fig. 4(b-1 and b-2) yield $l = 0.7181$ and $u = 0.7191$. $\square$

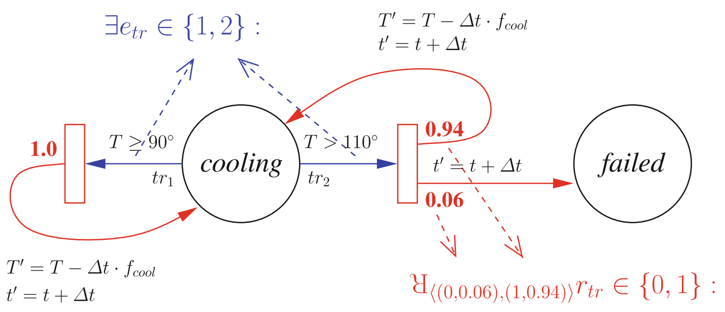
3.2.1 Encoding Bounded Reachability for SHA

CSSMT solving permits bounded model checking (BMC) of stochastic hybrid automata just the same way SMT facilitates it for hybrid automata (HA) devoid of stochasticity. The encodings of the transition relation are virtually identical to those established for HA [3, 12, 23], yet the alternation between non-deterministic and stochastic branching additionally has to be encoded by a corresponding alternation of $\exists$ and $\rotatebox[origin=c]{180}{R}$ quantifiers. The basic idea is illustrated in Fig. 5.

For computing the worst-case (in the sense that non-deterministic choices are resolved by a malicious adversary) probability of reaching a bad state in k steps, where bad states are encoded by a predicate Bad, this encoding is unwound to a formula Φ of the shape

$$\underbrace{\exists t_1 \rotatebox[origin=c]{180}{R}_d\, p_1 \exists t_2 \rotatebox[origin=c]{180}{R}_d\, p_2 \ldots \exists t_k \rotatebox[origin=c]{180}{R}_d\, p_k}_{\text{alternating non-determinist. and probabil. choices}} \ : \ \underbrace{\underbrace{\begin{pmatrix} Init(\vec{x}_0) \\ \wedge\, Trans(\vec{x}_0, \vec{x}_1) \\ \wedge\, Trans(\vec{x}_1, \vec{x}_2) \\ \wedge \ldots \\ \wedge\, Trans(\vec{x}_{k-1}, \vec{x}_k) \end{pmatrix}}_{k\text{-bounded reach set}} \wedge \underbrace{\begin{pmatrix} Bad(\vec{x}_0) \\ \vee\, Bad(\vec{x}_1) \\ \vee\, Bad(\vec{x}_2) \\ \vee \ldots \\ \vee\, Bad(\vec{x}_k) \end{pmatrix}}_{\text{hits bad state}}}_{\text{BMC}(k)},$$

where the k-fold alternating quantifier prefix reflects the alternation between non-deterministic and randomized choices inherent to the semantics of SHA and where the matrix is a conventional symbolic encoding of the k-step BMC problem. The


$\exists e_{tr} \in \{1,2\}$:
$T' = T - \Delta t \cdot f_{cool}$
$t' = t + \Delta t$
1.0
$T \geq 90°$
tr_1
cooling
$T > 110°$
tr_2
0.94
$t' = t + \Delta t$
0.06
failed
$T' = T - \Delta t \cdot f_{cool}$
$t' = t + \Delta t$
$\text{Я}_{\langle(0,0.06),(1,0.94)\rangle} r_{tr} \in \{0,1\}$:

source ∧	guard ∧	transition selection ∧	random distrib. ∧	action ∧	target
$\Big(cooling \wedge$	$(T \geq 90°) \wedge$	$(e_{tr} = 1) \wedge$	`true` $\wedge$	$\begin{pmatrix} (T' = T - \Delta t \cdot f_{cool}) \\ \wedge\ (t' = t + \Delta t) \end{pmatrix}$	$\wedge\, cooling' \Big) \vee$
$\Big(cooling \wedge$	$(T > 110°) \wedge$	$(e_{tr} = 2)$	$\wedge\ (r_{tr} = 0) \wedge$	$(t' = t + \Delta t)$	$\wedge\, failed' \Big) \vee$
$\Big(cooling \wedge$	$(T > 110°) \wedge$	$(e_{tr} = 2)$	$\wedge\ (r_{tr} = 1) \wedge$	$\begin{pmatrix} (T' = T - \Delta t \cdot f_{cool}) \\ \wedge\ (t' = t + \Delta t) \end{pmatrix}$	$\wedge\, cooling' \Big)$

Fig. 5 Principle of encoding stochastic hybrid automata: non-deterministic choice maps to existential quantification (*top left part* of the graphics), probabilistic choice to randomized quantification (*bottom right part* of the graphics), and the transition relation is encoded symbolically as in SMT-based BMC (table), yet adding the dependencies on choices (columns 3 and 4 in the formula)

quantifiers and the symbolic transition relation $Trans$ correspond to the respective objects in Fig. 5. Details of this encoding can be found in [15, 16, 37]. Its central property is that the satisfaction probability $Pr(\Phi)$ of the resulting formula is exactly identical to the worst-case probability of the encoded SHA reaching a bad state within k steps, such that CSSMT solving can be used for discharging the proof obligations arising from bounded probabilistic reachability problems. Verification problems of the form “can the the (worst-case under all adversary strategies resolving non-determinism) probability of reaching a bad state over a horizon of k steps be guaranteed to stay below a given safety target ε” are thus amenable to automatic analysis.

It should be noted that the above encoding yields extremely deep quantifier prefixes, as the alternation depth grows linearly in the number of steps of the probabilistic BMC problem. While this might seem to render (C)SSMT solving infeasible, adequate pruning rules in SSMT proof search permit to solve surprisingly large instances [37], speeding up solving by up to significantly more than ten orders of magnitude compared to naïve quantifier traversal.

4 Parameter Synthesis for Parametric Stochastic Hybrid Automata

Within the model of stochastic hybrid automata (SHA), we allowed for stochastic updates on the continuous as well as discrete variables using fixed probability distributions. This enables modeling of, among many other random phenomena, random component failures or data packet losses. Additionally, SHA contained non-determinism in terms of non-deterministic values and transitions. Extending SHA, within this section, we allow for an additional non-determinism in terms of a parametric dependence of the discrete probabilistic branching by replacing transition probabilities with parametric terms (t_i in Fig. 6). However, apart from this parametric non-determinism within the discrete probabilistic transitions, we require the system to be either deterministic or probabilistic, hence reducing the expressive power of SHA. Therefore, the resulting *parametric stochastic hybrid automata* (PSHA) feature a finite set of discrete locations (or modes), each of which comes decorated with a differential equation governing the dynamics of a vector of continuous variables while residing in that mode.

Modes change through instantaneous transitions guarded by conditions on the current values of the continuous variables, and may yield discontinuous (potentially stochastic) updates of the continuous variables. Aiming at simulation-based

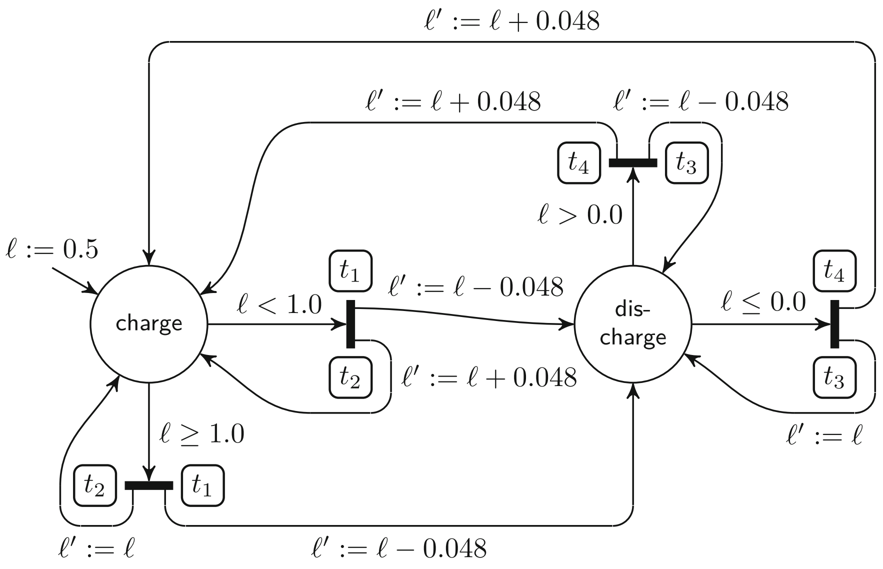


Fig. 6 PSHA model of a charging station. Modes are labeled with labels `charge` and `discharge` abbreviating ODE (not shown explicitly) representing corresponding dynamics over a continuous capacity ℓ. Modes can switch according to guarded transitions leading to a probabilistic branch. Probabilities are summarized as terms $t_1, \ldots, t_4$ indicating their parameter dependencies

evaluation methods as in Statistical Model Checking (SMC) [42], transition selection in this section is assumed to be deterministic, i.e., guard conditions at each mode are mutually exclusive or overlaps are resolved deterministically as, e.g., by the priority mechanisms in Simulink-Stateflow. To prevent non-determinism between possible time flows and transitions, we also assume that transitions are urgent, i.e., they are taken as soon as they are enabled (which furthermore renders mode invariants redundant). In addition to these mechanisms from deterministic hybrid automata, PSHA allow for the probabilistic selection of a transition variant based on a discrete random experiment. The probability distribution governing the random experiment is allowed to have a parametric dependence. Following the idea of Sproston [35, 36], the selected transition entails a randomized choice between transition variants according to a discrete probability distribution. The different transition variants can lead to different follow-up locations and different continuous successors, as depicted in Fig. 6, where the guard condition determining transition selection is depicted along the straight arrows leading to a potential branching annotated with probability terms denoting the random experiment.

To model the parametric dependence of PSHA, we allow the branching mode-transition probabilities to be terms over a set Θ of parameter names (t_i in Fig. 6). The viable parameter instances $\theta : \Theta \to \mathbb{R}$ are constrained by an arithmetic first-order predicate ϕ over Θ, defining their mutual relation. Let $\Phi = \{\theta : \Theta \to \mathbb{R} \mid \theta \models \phi\}$ denote the set of all viable parameterizations. Arithmetic terms over Θ are subject to the constraint that for all viable parameter valuations $\theta \models \phi$, the sum of outgoing probabilities assigned to each transition is 1, i.e., $\phi \implies \sum_{i=1}^{n} t_i(\theta) = 1$ holds for the probability terms t_i associated to each transition t. Note that the probability terms need not contain free variables from θ; non-parametric distributions are special cases of parametric distributions and do not require special treatment.

For the sake of formal analysis, we formalize the semantics of PSHA through a reduction to a parametric infinite-state Markov chain. For a PSHA with location set Λ and continuous variables $x_1, \ldots, x_D$, the states of the Markov chain are given by $\Sigma = \Lambda \times \mathbb{R}^D$ and the initial state distribution is inherited from the PSHA. Each state $\sigma = (l, \vec{x}) \in \Sigma$ gives rise to a parameter-dependent probability distribution of successor states σ':

$$p_\sigma(\sigma', \theta) = \begin{cases} t(\theta) & \text{if a transition } (\sigma, \sigma') \text{ labeled with probability term } t \text{ is enabled} \\ & \text{in } \sigma, \\ 1 & \text{if } \sigma' = (l, g(t)), \text{ where } g \text{ is a solution to the ODE associated to} \\ & l \in \Lambda \text{ with } g(0) = \vec{x}, \text{ and no transition is enabled in } (l, g(t')) \\ & \text{for any } t' \in [0, t[, \text{ and a transition is enabled in } \sigma' = (l, g(t)), \\ 0 & \text{otherwise.} \end{cases}$$

Given a parametric infinite-state Markov chain M with its initial state distribution given by a density $\iota : \Sigma \to \mathbb{R}_{\geq 0}$ and a parametric next-state probability mass function $p_\sigma : \Sigma \times \Phi \to [0, 1]$, the distribution associated to finite runs $\langle \sigma_0, \sigma_1, \dots, \sigma_k \rangle \in \Sigma^*$ given a parameter instance $\theta \in \Phi$ is given by the following probability mass function:

$$p_M(\langle \sigma_0, \sigma_1, \dots, \sigma_k \rangle; \theta) = \iota(\sigma_0) \cdot \prod_{i=0}^{k-1} p_{\sigma_i}(\sigma_{i+1}, \theta).$$

Note that θ can be vector valued, comprising multiple individual parameters.

Let $f : \Sigma \to \mathbb{R}$ be a scalar function on states, to be evaluated on the last state of a run and called the *reward* f of the run,[8] and let $k \in \mathbb{N}$. The *k-bounded expected reward for f in a parameter instance $\theta \in \Theta$* is

$$\mathcal{E}_{M,k}[f; \theta] = \int_{\Sigma^k} f(\sigma_{k-1}) p_M(\langle \sigma_0, \sigma_1, \dots, \sigma_{k-1} \rangle; \theta) \, \mathrm{d}\langle \sigma_0, \sigma_1, \dots, \sigma_{k-1} \rangle, \qquad (1)$$

where Σ^k denotes the sequences over Σ of length k. We will subsequently drop the index M in $\mathcal{E}_{M,k}$ and p_M whenever it is clear from the context. Although the finite nature of the parametric-dependent probabilistic branching would allow us to write the expectation in terms of a sum, we represent the expected value in form of an integral in Eq. (1) to illustrate the similarities to SHA and the general applicability of importance sampling (see below).

Rewards represent quantitative measures of the system's performance, and therefore mutual constraints on their values can be used for capturing design goals. The design problem we are thus facing is, given a vector $f_1, \dots, f_n : \Sigma \to \mathbb{R}$ of rewards in a parametric infinite-state Markov chain M, to ensure via adequate instantiation of the parameter that the expected rewards meet the design goal:

Definition 4 (*Parameter synthesis problem*) Let $f_1, \dots, f_n : \Sigma^k \to \mathbb{R}$ be a vector of rewards in a parametric Markov chain M and let C be a *design goal* in the form of a constraint on the expected rewards, i.e., an arithmetic predicate containing $f_1, \dots, f_n$ as free variables. A parameter instance $\theta : \Theta \to \mathbb{R}$ is *feasible (w.r.t. M and C)* iff

$$\theta \models \phi \quad \text{and} \quad [f_1 \mapsto \mathcal{E}(f_1; \theta), \dots, f_n \mapsto \mathcal{E}(f_n; \theta)] \models C.$$

The *multi-objective parameter synthesis problem* is to find a feasible parameter instance θ, if it exists, or to prove its absence otherwise.

Stated in words, a parameter instance θ is feasible w.r.t. ϕ and C iff the parameters are in the range defined by ϕ and the expected rewards resulting from the instantiation of M with θ meet the multi-objective C. Note that the aim is to find some parameter

[8]Due to the generality of the PSHA model, defining rewards exclusively on the final state is as expressive as defining them via functions on the whole run.

instance meeting our design goal; we are not considering determining the set of all suitable instantiations. That is, in contrast to the setting in the previous Sect. 3, we are here interested in a parameter value satisfying the constraints on multiple rewards rather than determining the probability of satisfaction for a given parameter instance. In fact, the constraint system of Definition 4 is indeed more general, as one can specify as objective a constraint system relating different expected values rather than just a required threshold on a single probability. For example, with the notation in Definition 4, it is possible to ask for a parameter instance, such that one expected value is larger than another expected value, both of which are allowed to depend on the parameter value under consideration. Note that such a problem cannot be stated using the CSSMT formalism of Sect. 3. On the other hand, the technology from Sect. 3 can deal with $1\frac{1}{2}$-player games, i.e., SHA involving both stochasticity and non-determinism, whereas we here deal with probabilistic systems only — albeit parametric ones.

4.1 Parameter Synthesis Using Symbolic Importance Sampling

Expected values aggregate contributions from many different states or trajectories as in Eq. (1). In particular, when each of the states contributes a different non-zero value to the overall expected values, an exact calculation of the expected value requires to evaluate all states or trajectories. As a result, the computation of the expected values can be the main bottleneck in finding a suitable parameter instance. We aim in the following at a statistical evaluation of the expected value, which scales with the number of samples used for such a statistical evaluation and therefore has the potential of producing results faster. As these statistical estimates of the expected values are merely arithmetic expressions, we then use these expressions to construct a constraint system which represents the parameter-dependencies of the expectations symbolically and can be solved using available constraint solvers, such as the iSAT solver [14]. However, due to the statistical sampling underlying the generation of the constraint system, these results are also only of statistical nature. That is, instead of finding a parameter instance for which we can rigorously guarantee that the constraint system is satisfied, we can only guarantee with a well-defined statistical confidence that the constraint system is likely to be satisfied. Similarly, we can only bound the probability that no parameter instance exists satisfying the constraint system in case of a negative satisfiability result.

In [18], we developed a scheme which uses a symbolic version of importance sampling in order to use a sampling-based strategy for estimating expected values while keeping track of the parametric dependence of these expectations, which we will review in the following. This technique combines statistical model-checking of a parameter-free instantiation of the parametric hybrid system with a symbolic variant of importance sampling in order to learn a symbolic model of the parameter dependency.

In order to introduce the general concept of importance sampling [39], we mostly abstract from our PSHA setting in this section, but highlight specifics to the PSHA setting when necessary. We instead assume that the parametric probability distribution of the random variable $x \in X$ is given in terms of a density function $p(\cdot; \theta)$ which depends on a vector θ of bounded real-valued parameters. Permissible values of θ are defined by a first-order constraint ϕ.

Given an arbitrary (bounded) function $f : X \to \mathbb{R}$, we are interested in estimating expected values of f under all parameter values $\theta \models \phi$. The expectation $\mathcal{E}[f; \theta]$ for reward f given parameter vector θ is

$$\mathcal{E}[f; \theta] = \int_X f(x) p(x; \theta) \, \mathrm{d}x \ . \tag{2}$$

Given a specific parameter instance θ^* and a process sampling x_i according to the distribution $p(\cdot; \theta^*)$, the expectation $\mathcal{E}[f; \theta^*]$ can be estimated by

$$\tilde{\mathcal{E}}[f; \theta^*] = \frac{1}{N} \sum_{i=1}^{N} f(x_i) \ , \tag{3}$$

which is the empirical mean of the sampled values of reward f. In our PSHA setting, a reasonable process for generating such samples x_i according to the distribution $p(\cdot; \theta^*)$ would be a simulator for non-parametric SHA, applied to the instance of the PSHA under investigation obtained by substituting concrete parameter values θ^* for the free parameters.

For sufficiently large N, we expect $\mathcal{E}[f; \theta^*] \approx \tilde{\mathcal{E}}[f; \theta^*]$ due to the law of large numbers. We can quantify the quality of the approximation in (3) using Hoeffding's inequality [24], provided that f has a bounded support $[a_f, b_f]$:

$$\begin{aligned} P\left(\mathcal{E}[f; \theta^*] - \tilde{\mathcal{E}}[f; \theta^*] \geq \varepsilon\right) &\leq \exp\left(-2 \frac{\varepsilon^2 N}{(b_f - a_f)^2}\right) \ , \\ P\left(\tilde{\mathcal{E}}[f; \theta^*] - \mathcal{E}[f; \theta^*] \geq \varepsilon\right) &\leq \exp\left(-2 \frac{\varepsilon^2 N}{(b_f - a_f)^2}\right) \ . \end{aligned} \tag{4}$$

Therefore, the empirical mean (3) yields a very reliable estimate of the actual expectation when the number of samples is large, with the accuracy given by (4).

As can be seen from Eq. (3), the estimate of the expected value depends only implicitly on the parameter of interest as the samples x_i are drawn from a fixed probability distribution using the concrete parameters θ^*. Consequently, one has to fix some parameter instance to generate the samples x_i, thereby losing the parametric dependence. To alleviate this problem, we one can use importance sampling using an arbitrary proposal distribution q, which does not depend on any parameter. Specifically, one can use the same sampling approach as (3) for (5), however, modifying the reward function:

$$\mathcal{E}[f;\theta] = \int_X f(x)\frac{p(x;\theta)}{q(x)}q(x)\,\mathrm{d}x \tag{5}$$

Using the same naïve Monte Carlo estimate yields the following empirical approximation to the expectation, including the parameter dependence on θ using N samples drawn from the substitute distribution q:

$$\tilde{\mathcal{E}}_q[f;\theta] = \frac{1}{N}\sum_{i=1}^{N} f(x_i)\frac{p(x_i;\theta)}{q(x_i)}\ . \tag{6}$$

Note that such a procedure can be used to obtain unbiased estimates of the expectation for both continuous probability distributions (densities) as well as discrete probability distributions (probability mass functions).

Doing so, however, requires being able to actually compute the quotient $\frac{p(x;\theta)}{q(x)} =: gain(x_i, \Theta)$ for each sample x_i. Whenever x_i is a trace in a PSHA, this can easily be achieved by taking

$$gain(x_i,\Theta) = \prod_{i=1}^{k}\left(\frac{t_i(\Theta)}{t_i(\Theta^*)}\right)^{\#t_i(x_i)}\ , \tag{7}$$

where $t_1, \ldots, t_k$ are the different parametric terms occurring in the automaton and $\#t_i(x_i)$ is the number of times the transition marked with t_i was taken in trace x_i. Note that $gain(x_i, \Theta)$ contains only the parameter vector Θ as free variables (all other entities are constants), such that (6) provides a *symbolic* expression for the parameter-dependency of $\mathcal{E}[f;\theta]$. For details concerning this automatically generated symbolic encoding, the interested reader may confer [18].

To synthesize a feasible parameter instance, we can generate an arithmetic constraint characterizing feasibility by plugging the symbolic parametric estimate (6) with the concrete gain term (7) into the condition for parameter feasibility from Definition 4, thereby adding an additional constraint for error control in the second line of the following equation. Here, $\epsilon(q_i, \delta, N)$ is an uncertainty term which captures the variability of the estimates as a function of the proposal distribution, the number of samples, and the confidence.

$$\begin{aligned} \theta \models \phi \quad &\text{and} \quad [f_1 \mapsto \mathcal{E}(f_1;\theta), \ldots, f_n \mapsto \mathcal{E}(f_n;\theta)] \models C \\ &\text{and} \quad \mathcal{E}(f_i;\theta)] \in [\tilde{\mathcal{E}}_{q_i}[f_i;\theta] - \epsilon(q_i,\delta,N), \tilde{\mathcal{E}}_{q_i}[f_i;\theta] + \epsilon(q_i,\delta,N)] \end{aligned} \tag{8}$$

Note that here, each of the expected values is replaced by the empirical estimate using samples drawn from a proposal distribution stemming from a parameter substitution Θ^*. As the resulting constraint system contains only constraints over arithmetic expressions, this system can directly be fed into a constraint solving engine such as iSAT [14], which is able to efficiently handle the polynomials of high degree stemming from the gain term (7). Due to the randomness involved in sampling, the estimates are themselves random variables. The result (a parameter instance or an infeasibility result) thus itself is subject to random fluctuations and we can guarantee the correctness of this result only with probability $\geq 1 - \delta$.

In order for this result to be valid, we have to determine $\epsilon(q_i, \delta, N)$ such that we can guarantee (with high probability)

$$\mathcal{E}(f_i;\theta) \in [\tilde{\mathcal{E}}_{q_i}[f_i;\theta] - \epsilon(q_i,\delta,N), \tilde{\mathcal{E}}_{q_i}[f_i;\theta] - \epsilon(q_i,\delta,N)] \ .$$

Importantly, due to the parameter dependence of the empirical estimates, one cannot use Hoeffding's inequality (4), but this statement has to hold uniformly across all θ satisfying ϕ.

Example 3 To illustrate this problem, consider the following example. Let $f(x) = \text{sign}(x - \pi)$ be a reward function on a continuous variable $x \in [0, 2\pi] \subset \mathbb{R}$. Further, let the parametric probability density $p(x;\theta) \propto (\sin(x\theta) + 1)$. When sampling x_i and calculating the empirical average reward, one can tune the corresponding Eq. (6) arbitrarily, such that the density $p(x;\theta)$ close to zero for all x_i with $f(x_i) = 1$ and close to one for all x_i with $f(x_i) = -1$ by appropriately choosing θ. However, the true expected reward is almost independent of θ. Importantly, for this choice of density, setting sampling points to zero or one is possible for arbitrarily many sampling points. Therefore, the empirical parametric expression does not necessarily converge to the true expectation with an increasing number of samples (as would be suggested by Hoeffding's inequality). To adjust for this effect, one has to account for the complexity of the parametric function. □

In fact, describing the effect of tuning parameters within such empirical expressions is one of the major research questions within the field of statistical learning theory (see [8, 41]), resulting in different complexity measures. For example, the sinusoidal function above has Vapnik-Chervonenkis dimension of infinity, indicating a very high complexity. By tuning the parameters after the data has been observed, a common phenomenon is called over-fitting, i.e., overly adapting the parameter to the data thereby introducing a larger error in the statistical estimate of the true value of the expected reward.

When using Eq. (8) together with a constraint solver as iSAT, we have to show the validity of the results, i.e., we have to show that the probability of the obtained result being wrong is bounded by a pre-specified value δ. To this end, we first consider the case that the constraint solver produces the result `UNSAT`. That is, the constraint solver cannot find a candidate solution θ^* satisfying the constraint system (8). The reason for such unsatisfiability can either rightfully lie within infeasibility of the synthesis problem itself, or can be erratic due to the statistical uncertainty within the estimation of the expected values.

`UNSAT` Case

In case the solver cannot find a candidate value for the parameter such that the constraints are satisfied, we would like to bound the probability for this statement being wrong, i.e., an artifact of the randomness in sampling. To bound this probability, we can examine the following events for arbitrary probability thresholds c:

$$E_1 : \min_\theta \mathcal{E}[f_i;\theta] < c \quad \text{and} \quad \min_\theta \tilde{\mathcal{E}}[f_i;\theta] \geq c + \epsilon \tag{9}$$

$$E_2 : \max_\theta \mathcal{E}[f_i;\theta] > c \quad \text{and} \quad \max_\theta \tilde{\mathcal{E}}[f_i;\theta] \leq c - \epsilon \tag{10}$$

Intuitively, we would like to bound the probability that we were not able to solve a slightly easier task $\tilde{\mathcal{E}}[f_i;\theta] \geq c + \epsilon$ while the original task is possible $\mathcal{E}[f_i;\theta] < c$.

Theorem 1 (Confidence for `UNSAT`)
Let $\epsilon = 2B_i\sqrt{-\frac{\log(\delta)}{N}}$ *and* $B_i = \max_{x,\theta} \frac{f_i(x)p(x;\theta)}{q_i}$ *be given. Then* $P(E_1) \leq \delta$ *and* $P(E_2) \leq \delta$.

Proof For E_1 the following holds:

$$\begin{aligned}
&P\left(\min_\theta \tilde{\mathcal{E}}[f_i;\theta] \geq \epsilon + c \ \wedge\ \min_\theta \mathcal{E}[f_i;\theta] < c\right)\\
&\leq P\left(\min_\theta \tilde{\mathcal{E}}[f_i;\theta] \geq \min_\theta \mathcal{E}[f_i;\theta] + \epsilon\right)\\
&\overset{\text{Jensen ineq.}}{\leq} P\left(\underbrace{\min_\theta \tilde{\mathcal{E}}[f_i;\theta]}_{=:g(x_1,\ldots,x_N)} - \mathcal{E}\left[\min_\theta \tilde{\mathcal{E}}[f_i;\theta]\right] \geq \epsilon\right)\\
&= P\left(g(x_1,\ldots,x_N) - \mathcal{E}\left[g(x_1,\ldots,x_N)\right] \geq \epsilon\right)\\
&\overset{\text{McDiarmid ineq.}}{\leq} \exp\left(-\frac{\epsilon^2 N}{4B_i^2}\right) = \delta;\ B_i = \max_{x,\theta} \frac{f_i(x)p(x;\theta)}{q_i(x)}
\end{aligned} \tag{11}$$

To apply McDiarmid's inequality [31] in Eq. (11), we used the following bounds:

$$\begin{aligned}
-\frac{2B}{N} &\leq \min_{\theta}\left\{\frac{1}{N}f(x_N;\theta) - \frac{1}{N}f(x'_N;\theta)\right\} \\
&\leq \min_{\theta}\left\{\frac{1}{N}\sum_{i}^{N-1} f\,(x_i;\theta) + \frac{1}{N}f(x_N;\theta)\right\} \\
&\qquad - \min_{\theta}\left\{\frac{1}{N}\sum_{i}^{N-1} f\,(x_i;\theta) + \frac{1}{N}f(x'_N;\theta)\right\} \\
&= \big(g(x_1,\ldots,x_N) - g(x_1,\ldots,x'_N)\big) \\
&\leq -\min_{\theta}\left\{\frac{1}{N}f(x'_N;\theta) - \frac{1}{N}f(x_N;\theta)\right\} \leq 2\frac{B}{N} \\
&\Rightarrow \left|\big(g(x_1,\ldots,x_N) - g(x_1,\ldots,x'_N)\big)\right| \leq 2\frac{B}{N}
\end{aligned} \tag{12}$$

In general, due to the concavity of the minimum, we have

$$\min(f-g) \leq \min(f) - \min(g) \leq -\min(-(f-g)) = \max(f-g) \tag{13}$$

For E_2 the proof is analogous replacing min with max. $\square$

SAT Case

Unfortunately, we are not able to construct a similar argument for the SAT case, as this would require calculating a complexity measure such as a Vapnik-Chernovenkis or Rademacher complexity for the function $\frac{f_i(x)}{q_i(x)}p(x;\cdot)$, for which the dependency on the parameter has to be analyzed to much more detail. However, if we construct the proposal distribution q_i by choosing a particular value of θ, e.g., $q_i(x) = p(x;\theta^*)$, we can use the standard Hoeffding inequality (4) to check whether this particular instance of parameter value θ^* satisfies the constraint system. That is, if we found that the constraint system (8) with the particular setting:

$$\tilde{\mathcal{E}}_{q_i}[f_i;\theta] = \frac{1}{N}\sum_{l} f_i(x_l) \text{ with } x_l \sim q_i = p(\cdot;\theta^*) \text{ and } \epsilon(q_i,\delta,N) = 2\sqrt{-\frac{\log(\delta)}{N}}$$

is satisfied for θ^*, then we know that the original constraint system is also satisfied with a probability $\geq 1-\delta$, due to Hoeffding's inequality.

Taken the results obtained so far, we have the following algorithm for checking a constraint system involving expected values of parametric probability distributions:

1. Select a particular parameter instance $\theta^* \models \phi$.
2. Draw N samples $\{x_i\}$, $i = 1,\ldots,N$ from the proposal distribution $q = p(\cdot;\theta^*)$.
3. Check the particular parameter instance for satisfaction of C using the empirical estimates $\tilde{\mathcal{E}}[f_i;\theta^*] = \frac{1}{N}\sum_k f_i(x_k)$ (see also (3)). To do so, we check if $\tilde{\mathcal{E}}[f_i;\theta^*] \pm 2\sqrt{-\frac{\log(\delta)}{N}}$ satisfies C. This step is referred to in Algorithm 1 as CHECKSAMPLES.

4. If the system is satisfied, we have found a feasible parameter instance θ^* with confidence $1-\delta$, i.e., the probability of violating the constraints on the expected rewards is smaller than δ. This is due to Hoeffding's inequality (4), which guarantees the high probability (see also [44] where the same method is applied).
5. If the system is unsatisfiable, we construct the empirical constraint system using (8).
6. Check the corresponding constraint system using iSAT once again.
7. If the system is unsatisfiable, we know with confidence $1-\delta$ that the original constraint system is unsatisfiable.

It could, however, happen that the first check yields unsatisfiable, while the second yields satisfiable, i.e., iSAT could neither verify the particular parameter instance θ^*, nor could refute existence of feasible parameter instances. In this case, the second check, however, generates another candidate parameter vector θ^{**} using the empirical parametric constraint system (8). With the newly obtained parameter instance, we can now re-iterate the algorithm until we either find a statistically valid solution or refute existence of feasible instances. Taken together, we obtain the iterative Algorithm 1.

Algorithm 1 Parameter Fitting by Symbolic Importance Sampling

```
function SYM- IMP($\phi$, $C$, confidence $\delta$, number of samples $N$, max. iterations $I$)
    $\delta^c \leftarrow \frac{\delta}{I}$; $\theta_0 \leftarrow$ SOLVECONSTRAINTSYSTEM($\phi$); $\varepsilon \leftarrow \sqrt{\frac{\log(\frac{1}{\delta^c})}{N}}2B$; $m \leftarrow 0$; $\hat{\phi}_0 \leftarrow \phi$
    while $m \leq I$ do
        $q \leftarrow p(\cdot;\theta_m)$
        $S = (x_1,\dots,x_N) \leftarrow$ DRAWSAMPLES($q$, $N$)        ▷ Simulate $N$ times w.
                                                                  ▷ parameterization $\theta_m$.
        if CHECKSAMPLES($S$, $\delta$, $\phi$, $C$) then
            return $\theta_m$        ▷ Found parameterization satisfying $C$ with prob. $\geq 1-\delta$
        else
            $\hat{\phi}_{m+1} \leftarrow \hat{\phi}_m \wedge \bigwedge_{i=1}^{n} \mathcal{E}(f_i;\theta) \in [\tilde{\mathcal{E}}_q[f_i;\theta] - \epsilon(q,\delta,N), \tilde{\mathcal{E}}_q[f_i;\theta] + \epsilon(q,\delta,N)]$
                                                                  ▷ Add samples to empirical system
            $\theta_{m+1} \leftarrow$ SOLVECONSTRAINTSYSTEM($\hat{\phi}_{m+1}$)
            if $\hat{\phi}_{m+1}$ is unsatisfiable then
                return Unsat        ▷ Original system is unsatisfiable with prob. $\geq 1-\delta$
            else $m \leftarrow m+1$
            end if
        end if
    end while
    return Unknown                                                ▷ Reached maximal iterations $I$
end function
```

In each iteration, we perform a hypothesis test by checking the satisfiability or unsatisfiability, adding to a maximum amount of hypothesis tests of I for each of both results. As the samples we use at one iteration are used in the next iteration as well by adding the corresponding empirical estimate as an additional constraint, the tests are not independent to each other. Note that we have to use a Bonferroni correction $\delta^c \leftarrow \frac{\delta}{I}$ to compensate for this dependency (see [32]). As we would like to have the final result to hold with confidence $1 - \delta$, the Bonferroni correction requires each of the hypothesis tests (there is a maximum of I) to give a valid result with at least $1 - \frac{\delta}{I}$, thereby guaranteeing that the overall probability of obtaining a valid result is bounded by $(1 - \frac{\delta}{I})^I \geq 1 - I\frac{\delta}{I} = 1 - \delta$.

Taken together, this implies that whenever the algorithm terminates with a definite result, this result is sound with a confidence $\geq 1 - \delta$. Hence, any parameter instance generated will actually satisfy the feasibility condition Definition 1 with probability $\geq 1 - \delta$. Likewise, an infeasibility result reported implies that the problem actually is infeasible with probability $\geq 1 - \delta$.

5 Conclusion

Addressing the quest for automatic analysis tools covering the state dynamics of hybrid discrete-continuous systems, we have over the past decade developed a rich set of constraint solvers facilitating their symbolic or mixed symbolic-numeric analysis, starting from the first practical SAT-modulo-theory solver for real arithmetic involving transcendental functions and thus going beyond the confined decidable fragments of arithmetic (iSAT, [14]) over the seamless integration of safe ODE enclosures in SAT-modulo-theory solving (odeSAT, [11]) to stochastic extensions of SAT-modulo-theory (SSMT and CSSMT, [15, 19]). These techniques permit key-press verification of bounded safety properties of the embedded system within its physical environment, whereby both qualitative, i.e., normative, and quantitative, stochastic models of system dynamics are supported. The related tools have been developed within the Transregional Collaborative Research Action SFB-TR 14 "Automatic Verification and Analysis of Complex Systems" (AVACS, www.avacs.org) and the Research Training Group DFG-GRK 1765: "System Correctness under Adverse Conditions" (SCARE, `scare.uni-oldenburg.de`) and some of them, like the iSAT tool, are freely available from the respective web sites.

Within this chapter, we have in particular elaborated on the most recent methods and tools from that series. These are able to, first, solve quantitative bounded reachability of stochastic hybrid systems involving both discrete and continuous non-determinism and stochasticity and, second, synthesize feasible parameters for probabilistic branching in such systems satisfying multi-objective design goals w.r.t. expected cost/rewards in parametric stochastic hybrid systems. The workhorses here are CSSMT solving (Continuous Stochastic Satisfiability Modulo Theory [19]) and a novel blend of statistical model checking and arithmetic constraint solving facilitated by a symbolic version of importance sampling [18]. We expect such combinations to have a much broader area of application, as they can be used for automatically mining from samples a formal model of rigorously controlled epistemological validity: the methods provide a learning scheme yielding a formal constraint model whose validity can be guaranteed up to a quantifiable confidence, as explained in Sect. 4. We are currently trying to exploit that latter fact for porting formal verification to safety-critical embedded software inherently devoid of a formal functional specification, like the computer vision components with their object classifiers trained by machine learning that are central to future automated driving functions.

References

1. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. Lecture Notes in Computer Science, vol. 736, pp. 209–229. Springer, New York (1993)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**, 3–34 (1995)
3. Audemard, G., Bozzano, M., Cimatti, A., Sebastiani, R.: Verifying industrial hybrid systems with MathSAT. ENTCS **89**(4) (2004)
4. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere et al. [7], chap. 26, pp. 825–885
5. Bellman, R.: A Markovian decision process. J. Math. Mech. **6**, 679–684 (1957)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS'99. Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer, New York (1999)
7. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
8. Bousquet, O., Boucheron, S., Lugosi, G.: Introduction to statistical learning theory. Advanced Lectures on Machine Learning, pp. 169–207. Springer, New York (2004)
9. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991)
10. Chaochen, Z., Ravn, A.P., Hansen, M.R.: An extended duration calculus for hybrid real-time systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. Lecture Notes in Computer Science, vol. 736, pp. 36–59. Springer, New York (1992)
11. Eggers, A., Fränzle, M., Herde, C.: SAT modulo ODE: a direct SAT approach to hybrid systems. In: Cha, S.S., Choi, J.Y., Kim, M., Lee, I., Viswanathan, M. (eds.) Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08). Lecture Notes in Computer Science, vol. 5311, pp. 171–185. Springer, New York (2008)
12. Fränzle, M., Herde, C.: Efficient proof engines for bounded model checking of hybrid systems. In: Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 04), Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier (2004)
13. Fränzle, M., Herde, C., Ratschan, S., Schubert, T., Teige, T.: Interval constraint solving using propositional SAT solving techniques. In: Proceedings of the CP 2006 First International Workshop on the Integration of SAT and CP Techniques, pp. 81–95 (2006)
14. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large nonlinear arithmetic constraint systems with complex boolean structure. JSAT **1**(3–4), 209–236 (2007)
15. Fränzle, M., Hermanns, H., Teige, T.: Stochastic satisfiability modulo theory: a novel technique for the analysis of probabilistic hybrid systems. In: Egerstedt, M., Mishra, B. (eds.) Proceedings of the 11th International Conference on Hybrid Systems: Computation and Control (HSCC'08). Lecture Notes in Computer Science (LNCS), vol. 4981, pp. 172–186. Springer, New York (2008)
16. Fränzle, M., Teige, T., Eggers, A.: Engineering constraint solvers for automatic analysis of probabilistic hybrid automata. J. Logic Algebr. Program. **79**, 436–466 (2010)

17. Fränzle, M., Hahn, E.M., Hermanns, H., Wolovick, N., Zhang, L.: Measurability and safety verification for stochastic hybrid systems. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, pp. 43–52. ACM (2011)
18. Fränzle, M., Gerwinn, S., Kröger, P., Abate, A., Katoen, J.: Multi-objective parameter synthesis in probabilistic hybrid systems. In: Sankaranarayanan, S., Vicario, E. (eds.) Formal Modeling and Analysis of Timed Systems - 13th International Conference, FORMATS 2015, Madrid, Spain, 2–4 September 2015, Proceedings. Lecture Notes in Computer Science, vol. 9268, pp. 93–107. Springer, New York (2015)
19. Gao, Y., Fränzle, M.: A solving procedure for stochastic satisfiability modulo theories with continuous domain. In: Campos, J., Haverkort, B.R. (eds.) Quantitative Evaluation of Systems, 12th International Conference, QEST 2015, Madrid, Spain, 1–3 September 2015, Proceedings. Lecture Notes in Computer Science, vol. 9259, pp. 295–311. Springer, New York (2015)
20. Granvilliers, L., Benhamou, F.: Realpaver: an interval solver using constraint satisfaction techniques. ACM Trans. Math. Softw. (TOMS) **32**(1), 138–156 (2006)
21. Groote, J.F., Koorn, J.W.C., van Vlijmen, S.F.M.: The safety guaranteeing system at station Hoorn-Kersenboogerd. In: Conference on Computer Assurance, pp. 57–68. National Institute of Standards and Technology (1995)
22. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M., Kurshan, R. (eds.) Verification of Digital and Hybrid Systems. NATO ASI Series F: Computer and Systems Sciences, vol. 170, pp. 265–292. Springer, New York (2000)
23. Herde, C., Eggers, A., Fränzle, M., Teige, T.: Analysis of hybrid systems using HySAT. In: The Third International Conference on Systems (ICONS 2008), pp. 196–201. IEEE Computer Society (2008)
24. Hoeffding, W.: Probability inequalities for sums of bounded random variables. J. Am. Stat. Assoc. **58**, 13–30 (1963)
25. Julius, A.A.: Approximate abstraction of stochastic hybrid automata. In: Hespanha, J.P., Tiwari, A. (eds.) Hybrid Systems: Computation and Control: 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, 29–31 March 2006. Proceedings. Lecture Notes in Computer Science, vol. 3927, pp. 318–332. Springer, New York (2006)
26. Lee, E.A., Zheng, H.: Operational semantics of hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC'05. Lecture Notes in Computer Science, vol. 3414. Springer, New York (2005)
27. Littman, M.L., Majercik, S.M., Pitassi, T.: Stochastic boolean satisfiability. J. Autom. Reason. **27**(3), 251–296 (2001)
28. Majercik, S.M.: Stochastic boolean satisfiability. In: Biere et al. [7], chap. 27, pp. 887–925
29. Majercik, S.M., Littman, M.L.: Maxplan: a new approach to probabilistic planning. AIPS **98**, 86–93 (1998)
30. Majercik, S.M., Littman, M.L.: Contingent planning under uncertainty via stochastic satisfiability. In: AAAI/IAAI, pp. 549–556 (1999)
31. McDiarmid, C.: On the method of bounded differences. Surv. Comb. **141**(1), 148–188 (1989)
32. Miller, R.G.: Simultaneous Statistical Inference. Springer, New York (1981)
33. Papadimitriou, C.H.: Games against nature. J. Comput. Syst. Sci. **31**(2), 288–301 (1985)
34. Ravn, A.P., Rischel, H.: Requirements capture for embedded real-time systems. In: Proceedings of IMACS-MCTS'91 Symposium on Modelling and Control of Technological Systems, Villeneuve d'Ascq, France, 7–10 May, vol. 2, pp. 147–152. IMACS (1991)
35. Sproston, J.: Decidable model checking of probabilistic hybrid automata. In: Joseph, M. (ed.) Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 1926, pp. 31–45. Springer, New York (2000)
36. Sproston, J.: Model checking for probabilistic timed and hybrid systems. Ph.D. thesis, University of Birmingham (2001)
37. Teige, T.: Stochastic satisfiability modulo theories: a symbolic technique for the analysis of probabilistic hybrid systems. Ph.D. thesis, Universität Oldenburg (2012)
38. Teige, T., Fränzle, M.: Stochastic satisfiability modulo theories for non-linear arithmetic. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 248–262. Springer, New York (2008)

39. Tokdar, S.T., Kass, R.E.: Importance sampling: a review. Wiley Interdiscip. Rev.: Comput. Stat. **2**(1), 54–60 (2010)
40. Tseitin, G.: On the complexity of derivations in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logics (1968)
41. Vapnik, V.N.: Statistical Learning Theory, vol. 1. Wiley, New York (1998)
42. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, 27–31 July 2002, Proceedings, pp. 223–235 (2002)
43. Zhang, L., She, Z., Ratschan, S., Hermanns, H., Hahn, E.M.: Safety verification for probabilistic hybrid systems. In: Proceedings of the 22nd International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 6174, pp. 196–211. Springer, New York (2010)
44. Zhang, Y., Sankaranarayanan, S., Somenzi, F.: Statistically sound verification and optimization for complex systems. In: Cassez, F., Raskin, J.F. (eds.) Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 8837, pp. 411–427. Springer, New York (2014)

MARS: A Toolchain for Modelling, Analysis and Verification of Hybrid Systems

Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao and Liang Zou

Abstract We introduce a toolchain MARS for Modelling, Analyzing and veRifying hybrid Systems we developed in the past years. Using MARS, we build executable models of hybrid systems using the industrial standard environment Simulink/Stateflow, which facilitates analysis by simulation. To complement simulation, formal verification of Simulink/Stateflow models is conducted in the toolchain via the following steps: first, we translate Simulink/Stateflow diagrams to Hybrid CSP (HCSP) processes by an automatic translator Sim2HCSP, where HCSP is an extension of CSP for formally modelling hybrid systems; second, to justify the translation, another automatic translator HCSP2Sim that translates from HCSP to Simulink is provided, so that the consistency between the original Simulink/Stateflow model and the translated HCSP formal model can be checked by co-simulation; then, the HCSP processes obtained in the first step are verified by an interactive Hybrid Hoare Logic (HHL) prover; during the verification, an invariant generator independent of the theorem prover for synthesizing invariants for differential equations and loops is needed. We

M. Chen (✉) · S. Wang · N. Zhan · L. Zou
State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, People's Republic of China
e-mail: chenms@ios.ac.cn

S. Wang
e-mail: wangsl@ios.ac.cn

N. Zhan
e-mail: znj@ios.ac.cn

L. Zou
e-mail: zoul@ios.ac.cn

X. Han · T. Tang
State Key Lab. of Rail Traffic Control and Safety, Beijing Jiaotong University, Beijing, People's Republic of China

M. Yang
Chinese Academy of Space Technology, Beijing, People's Republic of China

H. Zhao
School of Computer and Information Science, Southwest University, Chongqing, People's Republic of China

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_3

will demonstrate the toolchain by analysis and verification of a descent guidance control program of a lunar lander, which is a real-world industry example.

1 Introduction

Hybrid systems combine discrete controllers and continuous plants, and occur ubiquitously in safety-critical application areas such as transportation and avionics. To guarantee the correctness, formal techniques on modelling and verification of hybrid systems have been proposed [2, 20, 26, 28]. Besides, as a complementary activity to verification, several approaches have also been proposed for testing such systems [1, 3, 9]. However, the deep interactions between discrete and continuous components, and in addition, the complex continuous dynamics described by (non-linear) differential equations, make the formal analysis and verification of hybrid systems extremely difficult. Most existing work mentioned above can only deal with restricted systems, e.g., [2, 20] deal with dynamic and hybrid systems with a decidable reachability problem; [26] considered how to verify hybrid systems using simulation semantics, which cannot guarantee the correctness of hybrid systems in general because of the inherent incompleteness of simulation; while it is difficult to handle communication and parallelism using the approach in [28].

To develop reliable complicated hybrid systems, we propose the toolchain MARS for Modelling, Analyzing and veRifing hybrid systems. As shown in Fig. 1, the

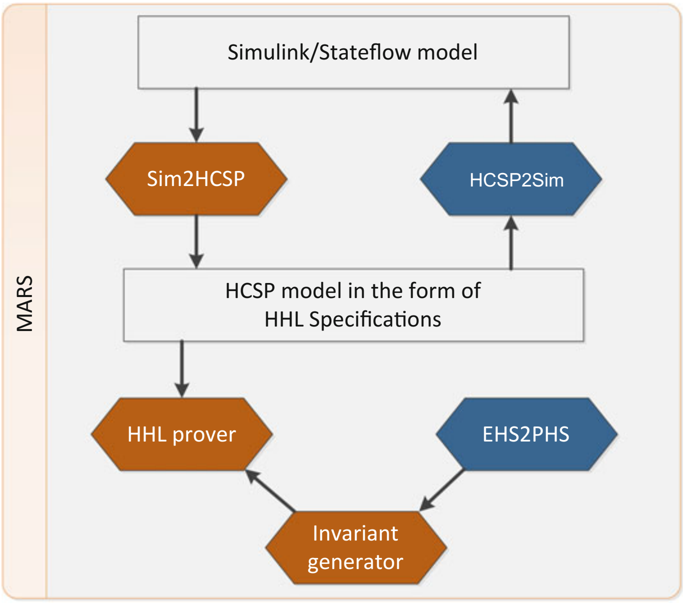


Fig. 1 Verification architecture

architecture of MARS is composed of three parts: a translator Sim2HCSP, an HHL prover, and an invariant generator. At the top level, we build executable models of hybrid systems in the graphical environment Simulink/Stateflow. As an industrial de-facto standard for designing embedded systems, Simulink/Stateflow facilitates the building of an executable model for a complicated system. Specifically, analysis and validation of a Simulink/Stateflow model can be conducted by simulation. However, simulation is inherently incomplete in coverage of system test cases and unsound due to numerical error. As a remedy, it deserves to further verify Simulink/Stateflow models in a formal verification tool.

In our approach, the translator Sim2HCSP is designed to translate Simulink/Stateflow models to HCSP [17, 39]. By extending CSP with differential equations, HCSP is a formal specification language for modelling hybrid systems, and meanwhile, it is the input language of the interactive HHL prover. By applying Sim2HCSP, the translation from Simulink/Stateflow to HCSP is fully automatic. Complementary to Sim2HCSP, an automatic inverse translator HCSP2Sim is implemented to justify its correctness. We use HCSP2Sim to translate the HCSP model resulting from Sim2HCSP back to Simulink, and check the consistency between the output Simulink/Stateflow model and the original Simulink/Stateflow model by co-simulation.

The HHL prover is then applied to verify the above HCSP models obtained from Sim2HCSP. The HHL prover is a theorem prover for Hybrid Hoare Logic (HHL) [21, 35]. As the input of the HHL prover, the HCSP models are written in the form of HHL specifications. Each HHL specification consists of an HCSP process, a pre-/post-condition that specifies the initial and terminal states of the process, and a history formula that records the whole execution history of the process, respectively. HHL defines a set of axioms and inference rules to deduce such a specification. Finally, by applying the HHL prover, the specification to be proved will be transformed into an equivalent set of logical formulas, which will be proved by applying axioms of corresponding logics in an interactive or automatic way.

To handle differential equations, we use the concept of *differential invariants* to characterize their properties without solving them [22, 29]. For computing differential invariants, we have implemented an independent invariant generator, which will be called during the verification in the HHL prover. The invariant generator integrates both the quantifier elimination and SOS (sum-of-squares) based methods for computing differential invariants of polynomial equations, and can also deal with non-polynomial systems by transformation techniques we proposed in [23], which is implemented as EHS2PHS in Fig. 1.

To evaluate MARS, we report our experience in using MARS on a case study in real industry, i.e. a descent guidance control program of a lunar lander, which is a closed-loop control system with non-linear differential equations.[1]

In our previous work [36], we studied the same example and verified it by combining several different verification techniques including simulation, bounded model

[1]The toolchain MARS and the verification of the lunar lander example can be found at http://lcs.ios.ac.cn/~znj/tools/MARS_v1.1.zip.

checking and theorem proving. In this chapter, we mainly focus on the tool implementation and integration, rather than on the case study itself as in [36]. The new contribution of this chapter is threefold:

- Firstly, we implement the reverse translator HCSP2Sim from HCSP to Simulink, to justify the correctness of the translation tool Sim2HCSP from Simulink to HCSP by co-simulation. This is not considered in the original version of Sim2HCSP presented in [40];
- Secondly, based on the invariant generation techniques proposed in [22, 23], we implement an invariant generator for differential equations and integrate it into the HHL prover. In [36], the invariants of related dynamics are synthesized manually. Besides, the tool EHS2PHS that abstracts a non-polynomial hybrid system by a polynomial one based on the technique in [23] is integrated to the invariant generator;
- Finally, we provide a seamless integration of all the tools on modelling, analysis and verification of hybrid systems as a toolchain MARS.

1.1 Related Work

There are some work on tools for formal verification of Simulink/Stateflow diagrams addressing both discrete and continuous blocks. In [5] Chen et al. proposed an approach that translates Simulink models to a real-time specification language and then validated the models via a generic theorem prover. However, their approach can only handle a special class of differential equations with closed form solutions, and cannot handle Stateflow diagrams. Tools based on numerical simulation or approximation are proposed. STRONG [12] performs bounded time reachability and safety verification for linear hybrid systems based on robust test generation and coverage. Breach [13] uses sensitivity analysis to compute approximate reachable sets and analyzes properties in the form of MITL based on numerical simulation. C2E2 [14] analyzes the discrete-continuous Stateflow models annotated with discrepancy functions by transforming them to hybrid automata, and then checks bounded time invariant properties of the models based on simulation.

There are some tools for verifying hybrid systems modelled by formal specification languages. The tool d/dt [4] provides reachability analysis and safety verification of hybrid systems with linear continuous dynamics and uncertain bounded input. iSAT-ODE [15] is a numerical SMT solver based on interval arithmetic that can conduct bounded model checking for hybrid systems. Flow* [6] computes over-approximations of the reachable sets of continuous dynamical and hybrid systems in a bounded time. Both iSAT-ODE and Flow* are able to handle non-polynomial ODEs (ordinary differential equations). Based on deductive method, the interactive theorem prover KeYmaera [30] (and its newly developed version KeYmaera X [16]) verifies hybrid systems specified using differential dynamic logic. These tools, however, are not directly applicable to Simulink/Stateflow models.

Organization. The rest of the chapter is organized as follows: Sect. 2 introduces the tool Sim2HCSP for translating Simulink/Stateflow models, as well as its inverse HCSP2Sim. Sections 3 and 4 introduce the HHL prover for verifying HCSP models and the invariant generator respectively. In each of the sections, the corresponding tool is demonstrated by the descent guidance control program of a lunar lander. Section 5 concludes the chapter.

2 Sim2HCSP Translator

In this section, we demonstrate a fully automatic translator *Sim2HCSP* [40, 42] that encodes Simulink/Stateflow diagrams into HCSP processes.

Simulink/Stateflow As an industrial de-facto standard, Simulink [31] is extensively used for modelling, simulating and analyzing multidomain dynamic and embedded systems. It provides a graphical block diagramming tool and a customizable set of block libraries for building executable models of embedded systems and their environments. A Simulink model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by sending messages through the wires between them. For an elementary bloc k, it basically gets input signals and computes the output signals assisted by a set of user-defined parameters to alter its functionalities. One typical parameter is the *sample time*, which defines how frequently the computation is taken. Two special values, 0 and -1, may be set for sample time, where 0 indicates that the block is used for simulating the physical environment and hence computes continuously, and -1 signifies that the sample time of the block is not determined yet, which will be determined by the sample times of the in-coming wires to the block. Thus, blocks are classified into two categories, i.e. *continuous* and *discrete*, according to their sample times.

As a toolbox integrated into Simulink, Stateflow offers the modelling capabilities of statecharts for reactive systems. It can be used to construct Simulink blocks, fed with Simulink inputs and produces Simulink outputs. A Stateflow diagram has a hierarchical structure, which can be an *AND diagram*, for which states are arranged in parallel and all of them become active whenever the diagram is activated; or an *OR diagram*, for which states are connected with transitions and only one of them becomes active when the diagram is activated. A Stateflow diagram consists of an alphabet of events and variables, a finite set of states, and transition networks.

Hybrid CSP Hybrid CSP (HCSP) [17, 39] is a formal modelling language for hybrid systems which extends CSP [18] by introducing differential equations, time constructs, and interrupts. In HCSP, exchanging data among processes is solely described by communications, and no shared variable is allowed between different processes in parallel. We denote by *dVar* and *cVar* the countable set of discrete and continuous variables respectively, and by *Chan* ranged over ch, ch_1, ..., the countable set of channels. The syntax of HCSP is given as follows:

$$\begin{aligned} P \ &\widehat{=}\ \text{skip} \mid x := e \mid ch?x \mid ch!e \mid P; Q \mid B \rightarrow P \mid P \sqcup Q \mid P^* \\ &\quad \mid \langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle \mid \langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq \talloblong_{i \in I} (io_i \rightarrow Q_i) \\ S \ &\widehat{=}\ P \mid S \| S \end{aligned}$$

Here ch, $ch_i \in Chan$, io_i stands for a communication event, i.e. either $ch_i?x$ or $ch_i!e$, $x \in dVar \cup cVar$, $s \in cVar$, B and e are Boolean and arithmetic expressions respectively, P, Q, Q_i are sequential processes, and S stands for a system, i.e. an HCSP process.

The intended meaning of the individual constructs is explained as follows:

- skip terminates immediately having no effect on variables; and $x := e$ assigns the value of expression e to x and then terminates.
- $ch?x$ receives a value along channel ch and assigns it to x, and $ch!e$ sends the value of e along ch. A communication takes place as soon as both the sending and the receiving parties are ready, and may cause one side to wait.
- The sequential composition $P; Q$ behaves as P first, and if it terminates, as Q afterwards.
- The conditional $B \rightarrow P$ behaves as P if B is true, and otherwise it terminates immediately.
- The internal choice $P \sqcup Q$ behaves as either P or Q, and the choice is made randomly by the system.
- The repetition P^* executes P for some finite number of times.
- $\langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle$ is the continuous evolution statement. It forces the vector s of real variables to evolve continuously according to the differential equations $\mathcal{F}$ as long as the Boolean expression B, which defines the *domain of* s, holds, and terminates when B turns false. For hybrid automata, non-determinism occurs when both the domain of the continuous evolution and the jump condition are satisfied, i.e. it can choose to stay in the continuous evolution, or leave it by making a discrete transition. In HCSP, there is no such non-determinism.
- $\langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq \talloblong_{i \in I} (io_i \rightarrow Q_i)$ behaves like the continuous $\langle \mathscr{F}(\dot{s}, s) = 0 \& B \rangle$, except that it is preempted as soon as one of the communications io_i takes place. That is followed by the respective Q_i. Notice that, if the continuous terminates before a communication among $\{io_i\}_{i \in I}$ occurs, then the process terminates immediately without waiting for communication. When multiple communications from $\{io_i\}_{i \in I}$ get ready simultaneously before the others, an internal choice among these ready communications occur.
- $S_1 \| S_2$ behaves as if S_1 and S_2 run independently except that all communications along the common channels connecting S_1 and S_2 are to be synchronized.

Sim2HCSP Translator Given a Simulink/Stateflow model, Sim2HCSP translates its Simulink and Stateflow parts separately. With the approach in [40], the Simulink part is translated into a set of HCSP processes, while using the approach in [42], the Stateflow part is translated into another set of HCSP processes. Then, these HCSP processes are composed in parallel to form the whole model of the system. The

Simulink and Stateflow diagrams in parallel transmit data or events via communications. Please refer to [40, 42] for details. Sim2HCSP takes Simulink/Stateflow models (in xml format, which is generated by a Matlab script) as input, and outputs several files as the definitions for the corresponding HCSP processes, which contain three files for defining variables, processes, and assertions for the Simulink part, and the same three files for each Stateflow diagram within the Stateflow part.

We demonstrate the translation approach by a scenario originating from the descent guidance control program of a lunar lander, which actually provides a specific sampled-data control system composed of the physical plant and the embedded control program.

Example 1 (running example) The guidance control program is built as a Simulink diagram in Fig. 2, which includes three parts: updating mass m, calculating acceleration aIC, and calculating thrust F_c. The sample time of all blocks is fixed as 0.128s, i.e. the period of the guidance program. In Fig. 2, block m_in reads mass m from the continuous plant (modelled as the Simulink diagram in Fig. 3) periodically, block Fc is used to calculate thrust F_c, and the rest are used to calculate acceleration aIC. In particular, there are two inputs for block Fc: the first is the acceleration aIC, which is defined as

$$-0.01(F_c/m - gM) - 0.6(v - vslw) + gM$$

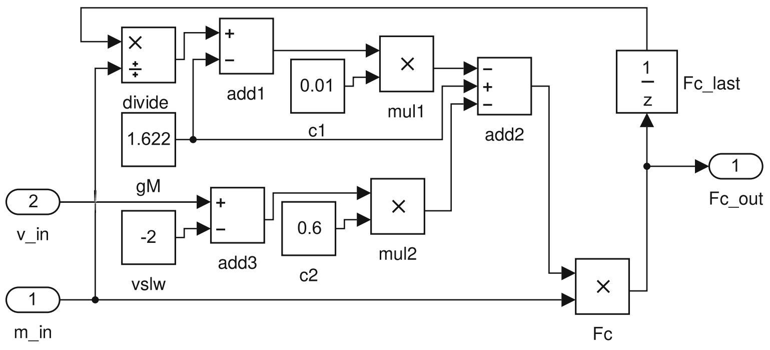


Fig. 2 Simulink diagram of the guidance program for the slow descent phase

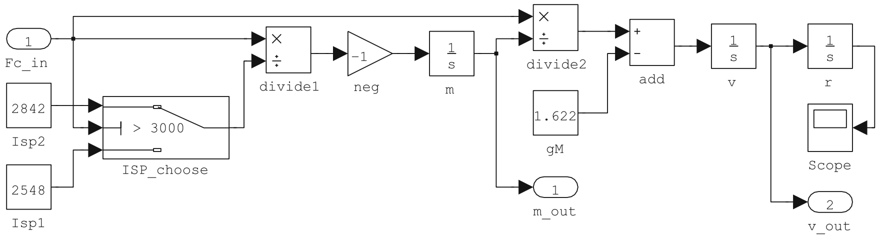


Fig. 3 The Simulink diagram of the dynamics for the slow descent phase

as shown in the diagram; the second is the mass m, and F_c is then defined as the product of aIC and m. The details of the guidance program can be found in [36].

The lander's dynamics is mathematically represented by

$$\begin{cases} \dot{r} &= v \\ \dot{v} &= \frac{F_c}{m} - gM \\ \dot{m} &= -\frac{F_c}{Isp} \\ \dot{F_c} &= 0 \end{cases} \tag{1}$$

where

- r, v and m denote the altitude (relative to lunar surface), vertical velocity and mass of the lunar lander, respectively;
- F_c is the thrust imposed on the lander, which is a constant in each sampling period of length 0.128 s;
- $gM = 1.622$ m/s^2 is the magnitude of the gravitational acceleration on the moon;
- Isp denotes the *specific impulse*[2] of the lander's thrust engine. It has two possible values depending on the values of F_c. When F_c is less or equal than 3000 N, $Isp = 2548$ N s/kg, and otherwise, $Isp = 2842$ N s/kg. For simplicity, we use Isp_1 and Isp_2 to represent the two values of the impulse, and meanwhile, use ODE_1 and ODE_2 to represent the two differential equations corresponding to Isp_1 and Isp_2 as defined by (1) respectively.

The physical dynamics in (1) is modelled by the diagram shown in Fig. 3, where the threshold of block ISP_choose is 3000, meaning that it outputs 2842 as the value of Isp when F_c is greater than 3000 and 2548 otherwise. The initial values of m, v, and r ($m = 1250$ kg, $r = 30$ m, $v = -2$ m/s) are specified as initial values of the *integrator* blocks m, v, and r respectively. Specifically, an integrator block outputs its initial value at the beginning and the integration of the input signal afterwards.

The safety property we want to prove for the lunar lander system is **Safety** $|v - vslw| \leq \varepsilon$, where $\varepsilon = 0.05$ m/s is the tolerance of fluctuation of v around the target $vslw = -2$ m/s.

The simulation result w.r.t the velocity v is illustrated in Fig. 4. It is shown that the velocity of the lander is kept between -2 and -1.9999 m/s, which corresponds to the safety property we proposed above.

Then the manually constructed Simulink model is translated into annotated HCSP using the tool Sim2HCSP, which employs the HCSP pattern

```
definition P :: proc where
"P == PC_Init; PD_Init; t:=0; (PC_Diff;t:=0;PD_Rep)*"
```

In process P, PC_Init and PD_Init are initialization procedures for the continuous dynamics and the guidance program respectively; PC_Diff models the continuous

[2]Specific impulse is a physical quantity describing the efficiency of rocket engines. It equals the thrust produced per unit mass of propellant burned per second.

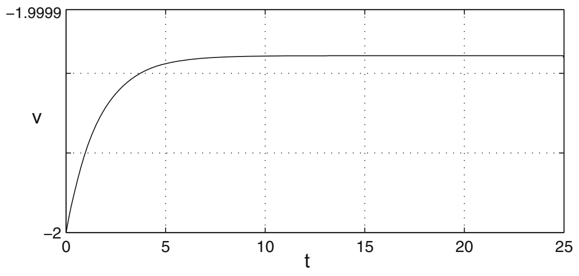


Fig. 4 The original simulation result

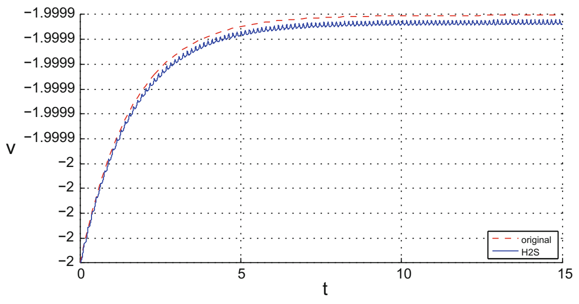


Fig. 5 The co-simulation result

dynamics given by (1) within a period of 0.128 s; PD_Rep calculates thrust F_c according to

$$F_c' := -0.01(F_c - m \cdot gM) - 0.6(v - vslw)m + m \cdot gM \tag{2}$$

for the next sampling cycle; variable t denotes the elapsed time in each sampling cycle. Hence, process P is initialized at the beginning by PC_Init and PD_Init, and behaves as a repetition of dynamics PC_Diff and computation PD_Rep afterwards.

Consistency Checking by Co-simulation To justify the correctness of the translation above, we provide a method to check the consistency between the original Simulink model and the generated HCSP formal model. This is done with the help of a tool called *HCSP2Sim* [7], an inverse decoding from HCSP back into Simulink. The translator HCSP2Sim takes as input an HCSP process transformed directly from the HCSP model generated by Sim2HCSP, and generates a Simulink graphical model in the mdl format automatically as output. Figure 5 illustrates the co-simulation result, where the evolution of the lander's velocity v in the original Simulink model is shown as the red dash line[3] and the one for the inversely translated Simulink model as the blue line. The co-simulation result shows that the translation loop keeps the behaviour of the system consistently. However, as also shown by the result, there exists a gap between the red and blue lines. This is the inevitable consequence of introducing some

[3]Identical to the line in Fig. 4.

necessary *delay* blocks in the translation from HCSP to Simulink, to prevent the *zeno*[4] phenomena while keeping the well-composed translation architecture. Nevertheless, absolute magnitude of the gap can be reduced by means of narrowing the simulation time step to an acceptable slot. In such way, a more precise co-simulation can be conducted. As an additional byproduct, the inverse translation also provides people with the ability to simulate an abstract formal model and see how the system behaves immediately and intuitively.

3 HHL Prover

This section presents the HHL prover for reasoning about HCSP models, and before that, gives a brief introduction of the Hybrid Hoare Logic (HHL) based on which the prover is implemented.

Hybrid Hoare Logic For verifying the behavior of HCSP processes, a deductive calculus called Hybrid Hoare Logic (HHL) is proposed in [21]. Given a process P, the specification $\{Pre\}P\{Post;\ \ HF\}$ is defined, where *Pre* and *Post* are first-order logic (FOL) formulas for specifying the pre-/post-conditions holding at the beginning and termination of P, and *HF* is a duration calculus (DC) [37, 38] formula for specifying the history throughout the whole execution of P. Here DC is an interval logic for describing real-time systems. In particular, as used below in the paper, ℓ is a temporal variable denoting the length of the considered interval, and $\lceil S \rceil$ for some FOL formula S means that S holds everywhere in the considered interval.

In HHL, for each HCSP construct, a set of inference rules are given for deducing its specifications. Below we explain the rule for the continuous evolution $\langle \mathscr{F}(\dot{s}, s) = 0\&B\rangle$. Instead of explicit solutions, the concept of *differential invariant* [22, 29] is used to characterize the behavior of the corresponding differential equations. As shown by the following rule, a differential invariant *Inv* needs to be annotated in the specification:

$$\frac{\begin{array}{c} Init \rightarrow Inv \quad (Inv, \mathcal{F}) \rightarrow Inv \quad p \wedge close(Inv) \wedge close(\neg B) \rightarrow q \\ l = 0 \vee \lceil close(Inv) \wedge p \wedge close(B) \rceil \rightarrow G \end{array}}{\{Init \wedge p\}\ \langle \mathscr{F}(\dot{s}, s) = 0\&Inv\&B\rangle\ \{q; G\}}$$

where *Init* specifies the initial state for s, p for other variables rather than s (thus will not change during the evolution), and function $close(\cdot)$ extends the domain by the corresponding formula to include the boundary; $(Inv, \mathcal{F})$ represents the formula describing the post-states of $\mathcal{F}$ executing from a state satisfying *Inv*. Consider the hypothesis, the FOL formula in the first line indicates that *Inv* is indeed a sufficiently strong invariant, i.e. it is satisfied by the initial state, preserved by the continuous

[4]A sequence of infinitely many computations that take finite time.

evolution, and strong enough to guarantee the postcondition; the DC formula in the second line indicates that the evolution terminates immediately (specified by $l = 0$), or otherwise, if the evolution takes more than zero time, then the closure of invariant *Inv*, the precondition p (related to discrete variables) and the closure of domain B hold everywhere throughout the whole execution. We have proved the soundness of the rule, and thus the proof of the specification of the continuous evolution will be reduced to an equivalent differential invariant generation problem: if *Inv* exists such that it satisfies the conditions in the hypothesis, then the original specification is proved.

The HHL Prover The interactive theorem prover *HHL prover*, as illustrated by Fig. 1, is implemented in Isabelle/HOL to mechanize the HHL framework and has been applied for verifying practical hybrid systems [36, 41]. The prover encodes the HHL framework in a deep style: the HCSP processes and the two assertion languages (i.e. FOL and DC) are defined by respective new datatypes, and in consequence, the inference system of HCSP (i.e. HHL), the deductive systems of FOL and DC are defined as new axioms, of Isabelle/HOL respectively. In the HHL prover, a set of verification conditions for HHL specifications are generated first by applying HHL inference rules, and then these conditions are proved by applying the FOL and DC deductive rules. Most of the proofs are done interactively. To improve this, we define a conversion function from our FOL formulas to HOL formulas and thus the existing proof tactics of Isabelle/HOL are applicable. For example, the powerful *sledgehammer* that integrates third-party SMT solvers such as Z3 [11] can be applied to prove FOL formulas in the HHL prover.

When the specification to be proved contains unknown differential invariants, some verification conditions related to the invariants remain unproved in HHL prover. For such cases, the prover needs to call external provers, e.g. the invariant generator in MARS, for solving the invariants. This will be explained in detail in the next section.

Example 2 (running example) In Sect. 2, by applying Sim2HCSP, we get the HCSP process P for the lunar lander example. In order to meet the design requirement of the control program, we need to prove the following specification for it:

```
{True} P {|v-vlsw|<=0.05; (l=0)|high(|v-vlsw|<=0.05)}
```

where *high* corresponds to the $\lceil\,\rceil$ operator in DC. The specification indicates that the slow descent phase satisfies the safety property, i.e., the difference between the velocity v and the target velocity *vlsw* is always at most 0.05. By applying HHL prover, the specification is finally reduced to the following five unsolved constraints for the differential invariants of P:

```
lemma cons1: "(t<=0.128) & (t>=0) & Inv |- |v-vlsw|<=0.05"
lemma cons2: "(v=-2) & (m=1250) & (Fc=2027.5)
  & (t=0) |- Inv"
lemma cons3: "(t= 0.128) & Inv
  |- substF([(t,0)], substF([(Fc,
      -0.01*(Fc-1.622*m) - 0.6*(v+2)*m + 1.622*m)],Inv))"
```

```
lemma cons4: "exeFlow(''v, m, r, t'',
  ''(Fc/m) - 1.622, -(Fc/2548), v, 1'',t < 0.128,Inv) |- Inv"
lemma cons5: "exeFlow(''v, m, r, t'',
  ''(Fc/m) - 1.622, -(Fc/2842), v, 1'',t < 0.128,Inv) |- Inv"
```

The intuitive explanation of the constraints is: during each period of length 0.128 s, the invariant *Inv* is sufficiently strong to deduce the safety property (*cons*1), the initial state satisfies *Inv* (*cons*2), the computation, and the continuous evolution governed by the two differential equations of P, preserve *Inv* respectively (*cons*3, *cons*4 and *cons*5). In the above constraints, function *exeFlow*(*ode*, f) for given equation *ode* and precondition f returns the postcondition after executing the continuous flow represented by *ode* from a state satisfying f. In the next section, we will show how to apply an external invariant generator to handle these constraints.

4 Invariant Generator

To prove the invariant related subgoals during the verification in the HHL prover, we need to call an external *invariant generator* from the HHL prover. The invariant generator of MARS provides two approaches to synthesizing invariants, i.e., quantifier elimination (QE) based and SOS based. Before introducing the invariant generator, we explain how to invoke an external prover in Isabelle.

4.1 Isabelle Oracle

Isabelle provides the oracle mechanism to use new decision procedures not based on its inference kernel. Listing 1 defines the oracle to decide invariant related constraints. Function *trans_allCons* translates an invariant constraint in the form of FOL formulas into the string representation expected by the solver. The core function *decide* takes a string representation of the invariant constraints and passes it to the script program implementing the invariant generator, and then returns true if an invariant exists such that the constraints are satisfied, or false otherwise. These two functions are then combined into the oracle *inv_oracle*, which verifies an input invariant constraint using *decide*, and outputs it as a theorem of Isabelle without any change if it is certified. Finally, to be used for Isabelle proofs, the oracle *inv_oracle* is wrapped into a tactic *inv_oracle_tac* and then a new method *inv_oracle* is created based on this tactic.

```
ML{*
fun trans_allCons t = ...
fun decide p = "$InvGen/script.sh "^"\""^p^"\""
  |> Isabelle_System.bash_output
  |> fst
  |> isTrue;*}
oracle inv_oracle = {* fn ct =>
```

```
  if decide (trans_allCons (Thm.term_of ct))
  then ct
  else error "Proof failed."*}
ML{*
val inv_oracle_tac =
  CSUBGOAL (fn (goal, i) =>
  (case try inv_oracle goal of
    NONE => no_tac
  | SOME thm => rtac thm i))*}
method_setup inv_oracle = {*
  Scan.succeed (K (Method.SIMPLE_METHOD' inv_oracle_tac))*}
```

Listing 1 The Oracle for deciding differential invariants

Depending on the different methods for computing differential invariants, we have implemented two oracles: *inv_oracle_qe* based on quantifier elimination, and *inv_oracle_sos* based on the SOS method. We will explain these methods in more detail in Sects. 4.2–4.5.

Example 3 (running example) By applying the oracle inv_oracle_sos, we have proved the conjunction of the unsolved five constraints presented in Example 2 as a lemma:

```
lemma allCons: "|- cons1 [&] cons2 [&] cons3 [&] cons4 [&] cons5"
apply (simp: add consi_def for all i)
apply inv_oracle_sos
done
```

At this state, by applying MARS, the verification of the safety for the lunar lander example thus is completed. Specifically, the manual proof script consists of approximately 300 lines and the verification is done within one minute on a 32-bit Linux computer with a 1.60GHz Intel Core-i5 processor and 4GB of RAM.

Next we present the invariant generator in detail.

4.2 Differential Invariant Generation

The basic idea of differential invariant generation is by using templates and constraint solving. For simplicity, we illustrate the idea on systems with a single ODE and no jumps. For such systems, the unresolved constraints as in Examples 2 and 3 would roughly be as follows:

(a) $\phi_{\text{pre}} \longrightarrow \phi_{\text{inv}}$;
(b) $\phi_{\text{inv}} \longrightarrow [\dot{x} = f]\phi_{\text{inv}}$;
(c) $\phi_{\text{inv}} \longrightarrow \phi_{\text{post}}$,

where

- (a) means that a certain precondition ϕ_{pre} implies the required invariant ϕ_{inv};
- (b) means that any trajectory of the ODE $\dot{x} = f$ starting from ϕ_{inv} will always satisfy ϕ_{inv}, that is, ϕ_{inv} is a differential invariant of $\dot{x} = f$;
- (c) means that the differential invariant ϕ_{inv} implies a certain postcondition ϕ_{post}.

For systems with different modes and jumps betweens these modes, as well as reset functions related to the jumps, additional constraints will be imposed, which are omitted here.

Example 4 In a more readable way, the five unresolved lemmas in Examples 2 and 3 impose the following constraints:

(C1) $t \le 0.128 \wedge t \ge 0 \wedge Inv \longrightarrow |v - vslw| \le 0.05$;
(C2) $v = -2 \wedge m = 1250 \wedge F_c = 2027.5 \wedge t = 0 \longrightarrow Inv$;
(C3) $t = 0.128 \wedge Inv \longrightarrow Inv(t \mapsto 0;\ F_c \mapsto F_c')$, with F_c' defined in (2);
(C4) Inv is the differential invariant of the constrained dynamical system

$$\langle ODE_1;\ 0 \le t \le 0.128 \ \wedge\ F_c \le 3000\rangle$$

(C5) Inv is also the differential invariant of the constrained dynamical system

$$\langle ODE_2;\ 0 \le t \le 0.128 \ \wedge\ F_c > 3000\rangle$$

where ODE_1 and ODE_2 are the dynamics in (1) corresponding to Isp_1 and Isp_2 respectively.

If ϕ_{pre} and ϕ_{post} are polynomial formulas, and f is a polynomial vector field, then we can try to generate ϕ_{inv} by defining a polynomial template, i.e. a polynomial formula with undetermined parameters as an invariant candidate and then solving certain constraints to get the parameters. We have the following two approaches for generating constraints from (a)–(c) and getting the parameters:

(1) **QE-Based**: transform (a), (b) and (c) into first-order polynomial formulas as proposed in [22] and then apply quantifier-elimination (QE) [8] to the quantified conjunction of the transformed formulas to see if the parameters have solutions;
(2) **SOS-Based**: transform (a), (b) and (c) into sum-of-squares (SOS) constraints as proposed in [19] and then use an SDP (semi-definite programming) solver to solve the constraints to get the values of parameters.

The QE-approach is exact and more general, and in particular, the transformation of [22] is sound and complete, while the SOS approach is more efficient due to the use of numerical computation. We have implemented invariant generators based on both QE and SOS, and integrated them into the MARS tool chain. We will give more details about the two generators in Sects. 4.4 and 4.5 respectively.

When ϕ_{pre} and ϕ_{post} are non-polynomial formulas, or f is a non-polynomial vector field, we will use the abstraction approach proposed in the next subsection.

4.3 Abstraction of Elementary Hybrid Systems by Variable Transformation

In practice, HSs (hybrid systems) may contain elementary functions such as exp, ln, sin, cos, etc., called *Elementary Hybrid Systems* (EHSs). Due to the non-polynomial expressions which lead to undecidable arithmetic, verification of EHSs is very hard. Existing approaches based on partition of the state space or overapproximation of reachable sets suffer from state space explosion or inflation of numerical errors. In [23], we proposed a symbolic abstraction approach that reduces EHSs to polynomial hybrid systems (PHSs), by replacing all non-polynomial terms with newly introduced variables. Thus the verification of EHSs is reduced to the one of PHSs, enabling us to apply all the well-established verification techniques and tools for PHSs to EHSs. In this way, it is possible to avoid the limitations of many existing methods. We have implemented the above abstraction procedure as a tool EHS2PHS.

For example, the dynamics of the lunar lander involves non-polynomial expression, $\dot{v} = \frac{F_c}{m} - gM$, which is abstracted by the tool EHS2PHS based on a rule of variable transformation, i.e. $a = \frac{F_c}{m}$, where a happens to be the instant acceleration produced by the thrust F_c of the lander. The equivalently transformed polynomial system will then be delivered to the invariant generator.

4.4 QE-Based Invariant Generator

The invariant generator based on quantifier elimination is implemented in Mathematica as a Wolfram script. It can be accessed in Isabelle through the method *inv_oracle_qe* using command *apply inv_oracle_qe*. The generator takes two parameters as input: constraints ϕ_{allCons} to be solved from the Isabelle function *trans_allCons* as shown in Listing 1, as well as a positive integer n through the user interface. The parameter n is the order of polynomials which will be used to generate a parameterized polynomial invariant template based on variables X extracted from ϕ_{allCons}. The parameters in the invariant template is denoted as U and there is a user interface to set certain parameters in U to 0 in order to reduce the difficulty of quantifier elimination. There is a placeholder inv in ϕ_{allCons}, which will then be replaced by the generated invariant template.

Now ϕ_{allCons} is a conjunction of constraints like those shown in Example 4. Then constraints like (C4) are translated into polynomial formulas using the technique proposed in [22], and accordingly, ϕ_{allCons} is transformed into a conjunction of polynomial formulas, denoted by ϕ_{poly}. Use the default quantifier elimination function *Resolve* in Mathematica to eliminate all the quantifiers in $\exists U \forall X : \phi_{\text{poly}}$, and a result *True* or *False* will be returned. The invariant generator will then pass this result to Isabelle.

4.5 SOS-Based Invariant Generator

In order to avoid the high complexity of quantifier elimination algorithms, which takes doubly exponential time on real closed fields [10], an alternative is provided to synthesize invariants based on sum-of-squares (*SOS*) relaxation approach in the study of polynomial hybrid systems [19]. Given a bunch of unproven constraints derived from Isabelle, the SOS-based invariant generator first transforms them into a sequence of SOS-constraints w.r.t the user-defined invariant template, and then invokes semidefinite programming (*SDP*) [27, 34] to solve the parameterized polynomial invariant.

We continue the lunar lander example to demonstrate the use of the generator. Like the QE method, the SOS-based invariant generator can be triggered in Isabelle by an oracle called *inv_oracle_sos*, in which a terminal window is initially popped-up for the user to specify the upper bound of the polynomial degree d (we assume that the undetermined invariant *Inv* is a semialgebraic set of the form $PInv \leq 0$, where *PInv* is a parameterized polynomial with degree d); and then a Mathematica script *ScriptGenerator* is executed to generate an SOS-constraint model *sosInv.m* written as a script of the Matlab-based optimization tool Yalmip [24, 25]. For instance, the safety constraint (C1) which is equivalent to

$$t \geq 0 \wedge t \leq 0.128 \wedge (v < -2.05 \vee v > -1.95) \rightarrow PInv > 0,$$

is transformed to an SOS-constraint:

$$SOS(PInv - s_1 * t * (0.128 - t) - s_2 * (v + 1.95) * (v + 2.05) - eps)$$

where $SOS(f)$ indicates that the function f is a sum-of-squares polynomial, s_1 and s_2 are both SOS polynomials, and *eps* is a given positive constant denoting a margin introduced to avoid the errors of numerical computation in Matlab; to determine the parameters in *PInv*, s_1, and s_2, as well as parameters in the other constraints, the Yalmip script *sosInv.m* is then executed in Matlab and invokes the solver SDPT-3 [32, 33] to solve all the SOS-constraints; finally, another Mathematica script *InvChecker* is called to check and return the solving result back to Isabelle, namely *True* if the problem is successfully solved, or *False* otherwise. With $d = 6$, we get a result of *True* associated with the invariant shown in Fig. 6 (left part), and complete the proof of lemma *allCons* in Example 3 eventually.

In addition, once the SOS-based invariant generator is triggered by applying oracle *inv_oracle_sos* in Isabelle, all the procedures described above, except for the pop-up terminal, are transparent to users, i.e. no Matlab desktop or Mathematica frontend can be observed. Therefore in order to give an intuitive observation of the invariant, we provide an additional notebook file *InvChecker.nb* that can be executed in a Mathematica frontend to plot a graphical region of the generated invariant as depicted by Fig. 6 (right part). Besides, to avoid synthesizing a false invariant due to numerical computation errors, we can also integrate symbolic posterior checking

$$\begin{aligned}
&2.716877217+0.2881*t+1.6781*v-0.3244*a+0.1974*t^2\\
&-0.0274*t*v+0.1110*v^2+0.0133*t*a+0.4345*v*a\\
&+0.5502*a^2-0.1210*t^3-0.0575*t^2*v-0.1659*t*v^2\\
&-0.0169*v^3-0.1182*t^2*a-0.5511*t*v*a+0.1171*v^2*a\\
&-0.7916*t*a^2+0.1479*v*a^2-0.0728*a^3+0.0659*t^4\\
&-0.0552*t^3*v+0.1924*t^2*v^2+0.2271*t*v^3+0.0623*v^4\\
&+0.0517*t^3*a+0.1108*t^2*v*a+0.2281*t*v^2*a\\
&+0.0464*v^3*a+0.5376*t^2*a^2-0.1645*t*v*a^2\\
&+0.0220*v^2*a^2+0.0033*t*a^3-0.0107*v*a^3+0.0230*a^4\\
&-0.3817*t^5+0.2199*t^4*v+0.0200*t^3*v^2+0.2136*t^2*v^3\\
&-0.0824*t*v^4-0.1764*t^4*a-0.1554*t^3*v*a\\
&-0.4660*t^2*v^2*a-0.2303*t*v^3*a-0.0869*t^3*a^2\\
&-0.1280*t^2*v*a^2-0.1325*t*v^2*a^2-0.0484*t^2*a^3\\
&-0.0240*t*v*a^3-0.0619*t*a^4+0.3226*t^6+0.0936*t^5*v\\
&+0.2142*t^4*v^2+0.0581*t^3*v^3+0.1452*t^2*v^4\le 0
\end{aligned}$$

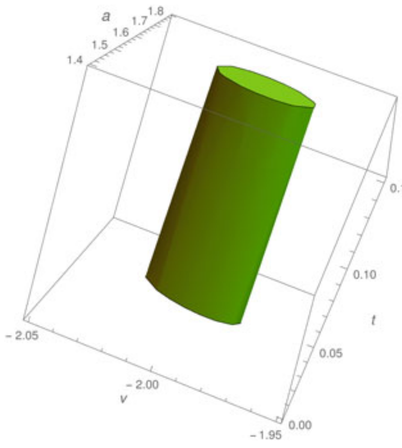


Fig. 6 The invariant generated by SOS relaxation with $d = 6$

of the generated invariants in *InvChecker.nb*, based on the symbolic computation packages provided in Mathematica.

5 Conclusion and Future Work

We presented a toolchain named MARS that links the modelling, analysis and verification of hybrid systems. The workflow of using MARS consists of the following phases: firstly, hybrid systems are modelled in the Simulink/Stateflow environment, which also facilitates model validation through numerical simulation; secondly, to overcome the limitations of simulation, the informal Simulink/Stateflow models are automatically transformed through the Sim2HCSP translator into formal models in the HCSP language; meanwhile, by an inverse translation from HCSP to Simulink models using the tool HCSP2Sim, and performing co-simulation, the consistency between the informal and formal models are justified; finally, the HCSP models can be verified preserving the given properties using the interactive HHL Prover, in which different schemes for automatic differential invariant generation are integrated, possibly with the support of EHS2PHS to abstract an EHS to a PHS first. We have discussed the details of the implementation of all components of MARS, and demonstrated how to use it through a real-life example of the slow descent control of a lunar lander.

As future work, we plan to improve MARS in the following aspects:the HHL prover needs improving its HHL verification framework and also its encoding in Isabelle/HOL so that more automation can be achieved for the proofs; the external invariant generators need to be enhanced with more efficient symbolic or hybrid numeric-symbolic computation techniques; the toolchain will be applied to other real-world case studies such as the modelling and verification of Chinese High-Speed Train Control System (CTCS); various component tools of MARS need to be more tightly integrated with a friendly user interface provided; and so on.

Acknowledgements The work is supported partly by "973 Program" under grant No. 2014CB-340701, by NSFC under grants 91418204 and 91118007, by CDZ project CAP (GZ 1023), and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

References

1. Aerts, A., Mousavi, M.R., Reniers, M.: A tool prototype for model-based testing of cyber-physical systems. In: Leucker, M., Rueda, C., Valencia, D.F. (eds.) ICTAC 2015, pp. 563–572. Springer International Publishing (2015)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. Lecture Notes in Computer Science, vol. 736, pp. 209–229. Springer, Berlin, Heidelberg (1993)
3. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011, pp. 254–257. Springer, Berlin, Heidelberg (2011)
4. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: CAV 2002. Lecture Notes in Computer Science, vol. 2404, pp. 365–370 (2002)
5. Chen, C., Dong, J.S., Sun, J.: A formal framework for modelling and validating Simulink diagrams. Form. Asp. Comput. **21**(5), 451–483 (2009)
6. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: CAV 2013. Lecture Notes in Computer Science, vol. 8044, pp. 258–263 (2013)
7. Chen, M., Ravn, A., Yang, M., Zhan, N., Zou, L.: A two-way path between formal and informal design of embedded systems. In: Proc. UTP 2016 (2016)
8. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Brakhage, H. (ed.) Automata Theory and Formal Languages. Lecture Notes in Computer Science, vol. 33, pp. 134–183. Springer, Berlin, Heidelberg (1975)
9. Dang, T., Nahhal, T.: Coverage-guided test generation for continuous and hybrid systems. Form. Methods Syst. Des. **34**(2), 183–213 (2009)
10. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. **5**(1–2), 29–35 (1988)
11. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS 2008. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin, Heidelberg (2008)
12. Deng, Y., Rajhans, A., Julius, A.A.: STRONG: a trajectory-based verification toolbox for hybrid systems. In: QEST 2013. Lecture Notes in Computer Science, vol. 8054, pp. 165–168 (2013)
13. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: CAV 2010. Lecture Notes in Computer Science, vol. 6174, pp. 167–170 (2010)
14. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for annotated Stateflow models. In: TACAS 2015. Lecture Notes in Computer Science, vol. 9035, pp. 68–82 (2015)
15. Eggers, A., Ramdani, N., Nedialkov, N., Fränzle, M.: Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In: SEFM 2011, pp. 172–187. Springer-Verlag, Berlin, Heidelberg (2011)
16. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. CADE **2015**, 527–538 (2015)

17. He, J.: From CSP to hybrid systems. In: A Classical Mind, Essays in Honour of C.A.R. Hoare, pp. 171–189. Prentice Hall International (UK) Ltd. (1994)
18. Hoare, C.: Communicating Sequential Processes, vol. 178. Prentice-hall Englewood Cliffs (1985)
19. Kong, H., He, F., Song, X., Hung, W.N., Gu, M.: Exponential-condition-based barrier certificate generation for safety verification of hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. Lecture Notes in Computer Science, vol. 8044, pp. 242–257. Springer, Berlin Heidelberg (2013)
20. Lafferriere, G., Pappas, G.J., Yovine, S.: Symbolic reachability computation for families of linear vector fields. J. Symb. Comput **32**(3), 231–253 (2001)
21. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. Lecture Notes in Computer Science, vol. 6461, pp. 1–15. Springer, Berlin, Heidelberg (2010)
22. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011, pp. 97–106. ACM, New York, NY, USA (2011)
23. Liu, J., Zhan, N., Zhao, H., Zou, L.: Abstraction of elementary hybrid systems by variable transformation. In: FM 2015. Lecture Notes in Computer Science, vol. 9109, pp. 360–377 (2015)
24. Löfberg, J.: YALMIP: a toolbox for modeling and optimization in MATLAB. In: Proceedings of the CACSD Conference. Taipei, Taiwan (2004). http://users.isy.liu.se/johanl/yalmip
25. Löfberg, J.: Pre- and post-processing sum-of-squares programs in practice. IEEE Trans. Autom. Control **54**(5), 1007–1011 (2009)
26. Manna, Z., Pnueli, A.: Verifying hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. Lecture Notes in Computer Science, vol. 736, pp. 4–35. Springer, Berlin, Heidelberg (1993)
27. Parrilo, P.A.: Semidefinite programming relaxations for semialgebraic problems. Math. Program. **96**(2), 293–320 (2003)
28. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Logic Comput. **20**(1), 309–352 (2010)
29. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. Lecture Notes in Computer Science, vol. 5123, pp. 176–189. Springer, Berlin, Heidelberg (2008)
30. Platzer, A., Quesel, J.D.: KeYmaera: a hybrid theorem prover for hybrid systems. In: IJCAR 2008. Lecture Notes in Computer Science, vol. 5195, pp. 171–178. Springer, Berlin, Heidelberg (2008)
31. Simulink User's Guide. http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf (2013)
32. Toh, K.C., Todd, M., Tütüncü, R.H.: SDPT3 – a MATLAB software package for semidefinite programming. Optim. Methods Softw. **11**, 545–581 (1999)
33. Tütüncü, R.H., Toh, K.C., Todd, M.J.: Solving semidefinite-quadratic-linear programs using SDPT3. Math. Program. **95**(2), 189–217 (2003)
34. Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Rev. **38**(1), 49–95 (1996)
35. Wang, S., Zhan, N., Zou, L.: An improved HHL prover: an interactive theorem prover for hybrid systems. In: ICFEM 2015. Lecture Notes in Computer Science, vol. 9407, pp. 382–399 (2015)
36. Zhao, H., Yang, M., Zhan, N., Gu, B., Zou, L., Chen, Y.: Formal verification of a descent guidance control program of a lunar lander. In: FM 2014. Lecture Notes in Computer Science, vol. 8442, pp. 733–748 (2014)
37. Zhou, C., Hansen, M.R.: Duration Calculus – A Formal Approach to Real-Time Systems. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin Heidelberg (2004)
38. Zhou, C., Hoare, C., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991)

39. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) Hybrid Systems III. Lecture Notes in Computer Science, vol. 1066, pp. 511–530. Springer, Berlin, Heidelberg (1996)
40. Zou, L., Zhan, N., Wang, S., Fränzle, M., Qin, S.: Verifying Simulink diagrams via a Hybrid Hoare Logic prover. EMSOFT **2013**, 1–10 (2013)
41. Zou, L., Lv, J., Wang, S., Zhan, N., Tang, T., Yuan, L., Liu, Y.: Verifying Chinese train control system under a combined scenario by theorem proving. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. Lecture Notes in Computer Science, vol. 8164, pp. 262–280. Springer, Berlin Heidelberg (2014)
42. Zou, L., Zhan, N., Wang, S., Fränzle, M.: Formal verification of Simulink/Stateflow diagrams. In: ATVA 2015. Lecture Notes in Computer Science, vol. 9346, pp. 464–481 (2015)

Part III
Correctness of Concurrent Algorithms

A Proof Method for Linearizability on TSO Architectures

John Derrick, Graeme Smith, Lindsay Groves and Brijesh Dongol

Abstract Linearizability is the standard correctness criterion for fine-grained non-atomic concurrent algorithms, and a variety of methods for verifying linearizability have been developed. However, most approaches to verifying linearizability assume a sequentially consistent memory model, which is not always realised in practice. In this chapter we study the use of linearizability on a *weak* memory model. Specifically we look at the TSO (Total Store Order) memory model, which is implemented in the x86 multicore architecture. A key component of the TSO architecture is the use of write buffers, which are used to store pending writes to memory. In this chapter, we explain how linearizability is defined on TSO, and how one can adapt a simulation-based proof method for use on TSO. Our central result is a proof method that simplifies simulation-based proofs of linearizability on TSO. The simplification involves constructing a coarse-grained abstraction as an intermediate specification between the abstract representation and the concurrent algorithm.

1 Introduction

Concurrency is here to stay. Furthermore, many systems and most multiprocessors use shared memory. The use of concurrent algorithms to optimise performance is likely to be important for some time in this scenario, where fine-grained algorithms implement single atomic operations as interleaved non-atomic decompositions. The

J. Derrick (✉)
Department of Computing, University of Sheffield, Sheffield, UK
e-mail: J.Derrick@sheffield.ac.uk

G. Smith
School of Information Technology and Electrical Engineering,
The University of Queensland, St Lucia, QL, Australia

L. Groves
School of Engineering and Computer Science, Victoria University of Wellington,
Wellington, New Zealand

B. Dongol
Department of Computer Science, Brunel University of London, London, UK

use of such algorithms is already common-place, implementing data structures such as stacks, queues, trees, etc., and they are now found in standard programming libraries. In order to fully exploit the potential concurrency, algorithms dispense with large-scale locking of data structures in the shared memory to prevent lengthy delays. This means that the shared data structure can be concurrently accessed by different processors executing possibly different operations. This offers speed-ups over algorithms that use large-scale locking mechanisms, however, this optimisation comes at a price – that of verifying their correctness.

There has been extensive work on correctness of fine-grained concurrent algorithms over the last few years [14], where *linearizability* [16] is the key criteria that is applied. This requires that fine-grained implementations of data structure operations appear as though they take effect "instantaneously at some point in time" between their invocation and response [16], thereby achieving the same effect as an atomic operation. However, the vast majority of work on linearizability assumes a particular memory model; specifically a *sequentially consistent* (SC) memory model, whereby memory instructions are executed by the hardware in the order specified by the program. Typical multicore systems communicate via shared memory and, to increase efficiency, use (local) write buffers. Whilst these *relaxed memory models* give greater scope for optimisation, sequential consistency is lost, and because memory accesses may be reordered in various ways it is even harder to reason about correctness. Typical multiprocessors that provide such weaker memory models include the x86 [21], Power [23] and ARM [1] multicore processor architectures.

In this chapter, we focus on the TSO (Total Store Order) model [23] which is implemented in the x86 architecture. We define a notion of linearizability for use on this architecture, called *TSO-linearizability* [11]. Verifying linearizability on a sequentially consistent memory model can be challenging even without the additional complexity that TSO introduces due to the reordering of the memory accesses. We describe how we can simplify the verification to reduce some of this complexity. We do this by observing that in many cases the proof obligations required of TSO-linearizability can be split into two: one aspect dealing with the fine-grained nature of the concurrent algorithm, and the other with the effect the local write buffers have on the shared memory.

We exploit this observation in our proof method, which uses a coarse-grained abstraction that lies between the abstract specification and the concurrent algorithm. The coarse-grained abstraction captures the semantics of the concurrent algorithm when there is no fine-grained interleaving of operations by different processes. Our simplified proof method then requires one set of proof obligations between the concurrent algorithm and the coarse-grained abstraction, and another set of proof obligations between the coarse-grained abstraction and the abstract description. The proof method, originally proposed in [12], is extended in this chapter to be less dependent on the form of the abstract specification, and hence more generally applicable.

We structure the chapter as follows. In Sect. 2 we introduce the standard definition of linearizability on SC architectures and present an existing proof method for it. In Sect. 3 we introduce the TSO model and formalise a notion of linearizability on TSO previously published in [11]. In Sect. 4 we show how to construct a coarse-grained abstraction. We define a transformation from the coarse-grained abstraction to the abstract one, which together with the results of Sect. 2 allows us to prove overall correctness of the concrete specification with respect to the abstract one. We then show how to apply the approach to a more complex example, the Chase-Lev work-stealing deque [6], in Sect. 5, before concluding in Sect. 6.

2 Linearizability

Linearizability [16] is widely regarded as the standard correctness criterion for concurrent objects. Given an abstract specification and a proposed implementation, the idea of linearizability is that any concurrent execution of the implementation must be consistent with *some* abstract execution of the specification.

> Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its return. This point is known as the *linearization point*.

This means that if two operations overlap, then they may take effect in any order from an abstract perspective, but otherwise they must take effect in the order in which they are invoked.

Since the original definition there has been considerable interest in deriving techniques for verifying linearizability [14]. These range from using shape analysis [2, 5] and separation logic [5] to rely-guarantee reasoning [25] and refinement-based simulation methods [13]. In particular, Derrick et al. have developed a refinement-based method for verifying linearizability [8–10, 20]. This approach is fully encoded in a theorem proving tool, KIV [19], and has been proved sound and complete — the proofs themselves being done within KIV.

Case study: Before providing a formal definition of linearizability we introduce our running example – a *spinlock* [3], which is a locking mechanism designed to avoid operating system overhead associated with process scheduling and context switching.

The abstract specification (given below in Z) simply describes a lock, with operations $Acquire_p$, $Release_p$ and $TryAcquire_p$ parameterised by the identifier of the process $p \in P$ performing the operation, where P is the set of all process identifiers. A global variable x represents the lock and is set to 0 when the lock is held by a thread, and 1 otherwise.

```
int x = 1;

acquire() {                      release( ) {        tryacquire( ) {
  a1  while(1) {                   r1   x = 1;          t1  lock;
  a2     lock;                   }                      t2  if (x==1) {
  a3     if (x==1) {                                    t3     x = 0;
  a4        x = 0;                                      t4     unlock;
  a5        unlock;                                     t5     return 1;
  a6        return;                                         }
         }                                              t6  unlock;
  a7     unlock;                                        t7  return 0;
  a8     while(x==0) {};                             }
      }
}
```

Fig. 1 Spinlock implementation

$$\begin{array}{|l}\hline \multicolumn{1}{l}{AS} \\ x : \{0,1\} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \multicolumn{1}{l}{ASInit} \\ AS \\ \hline x = 1 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{Acquire_p} \\ \Delta AS \\ \hline x = 1 \\ x' = 0 \\ \hline \end{array} \qquad \begin{array}{|l}\hline \multicolumn{1}{l}{Release_p} \\ \Delta AS \\ \hline x' = 1 \\ \hline \end{array} \qquad \begin{array}{|l}\hline \multicolumn{1}{l}{TryAcquire_p} \\ \Delta AS \\ out! : \{0,1\} \\ \hline \mathbf{if}\ x = 1 \\ \mathbf{then}\ x' = 0 \wedge out! = 1 \\ \mathbf{else}\ x' = x \wedge out! = 0 \\ \hline \end{array}$$

A typical implementation of spinlock (taken from [15]) is shown in Fig. 1, given as pseudo-code. Line numbers, `a1`, etc., are given to the left of the code, corresponding to the atomic steps of the operations. A thread trying to acquire the lock *spins*, i.e., waits in a loop, while repeatedly checking `x` for availability.

A terminating `acquire` operation will always succeed in acquiring the lock. It will lock the global memory[1] so that no other process can write to `x`. If, however, another thread has already acquired the lock (i.e., `x==0`) then it will unlock the global memory and spin, i.e., loop in the while-loop until it becomes free, before starting over. Otherwise, it acquires the lock by setting `x` to 0.

The operation `release` releases the lock by setting `x` to 1. The `tryacquire` operation differs from `acquire` in that it only makes one attempt to acquire the lock. If this attempt is successful it returns 1, otherwise it returns 0.

The point about this concurrent implementation is that it is *fine-grained*. That is, the operations `acquire`, etc., are not executed atomically, but the individual

[1]Locking the global memory is achieved by calling an atomic hardware instruction (in this case, a test-and-set). It should not be confused with acquiring the *software* lock of this case study by setting `x` to 0.

statements of different threads, `a1`, `r1`, etc., can interleave. Linearizability is a means to ask whether such an interleaved execution is in fact consistent with its atomic abstract counter-part.

2.1 A Formal Definition of Linearizability

Formally, linearizability is defined in terms of *histories*, which are sequences of *events* which can be invocations or returns of operations from a set I performed by a particular process from a set P. Invocations have an associated input from domain In, and returns have an output from domain Out. Both domains contain the value $\perp$ indicating no input or output. We therefore define:

$$\begin{aligned}&Event \mathrel{\widehat=} inv\langle\langle P \times I \times In\rangle\rangle \mid ret\langle\langle P \times I \times Out\rangle\rangle\\ &History \mathrel{\widehat=} \operatorname{seq} Event\end{aligned}$$

Notation: For a history h, $h = \langle head\ h\rangle ^\frown tail\ h$ (where $^\frown$ is sequence concatenation), $\#h$ is the *length* of the sequence, and $h(n)$ its nth element (for $n : 1..\#h$). Predicates $inv?(e)$ and $ret?(e)$ determine whether an event $e \in Event$ is an invoke or return, respectively. We let $e.i \in I$ denote the operation of an event e, $e.\pi \in P$ denote the process which performs e, and $e.v \in In \cup Out$ denote the input/output value. Two indices m and n match in history h (denoted $match(h, m, n)$) iff $0 < m < n \le \#h \wedge h(m).\pi = h(n).\pi \wedge h(m).i = h(n).i \wedge inv?(h(m)) \wedge ret?(h(n))$. □

As in [9], we let $mp(h, m, n)$ identify matching pairs of invocations and returns in history h. Its definition requires that $h(m)$ and $h(n)$ are an invocation and a return event, respectively, of the same operation, executed by the same process p. Additionally, it requires that there are no invocation or return events of p between positions m and n in h. That is:

$$mp(h, m, n) \mathrel{\widehat=} match(h, m, n) \wedge \forall k \bullet m < k < n \Rightarrow h(k).\pi \neq h(m).\pi$$

Since operations are atomic in an abstract specification, its histories are *sequential*, i.e., each operation invocation is followed immediately by its return. For example,

$$\begin{aligned}h_s \mathrel{\widehat=} \langle &inv(p, \texttt{acquire}, \perp), ret(p, \texttt{acquire}, \perp), inv(q, \texttt{tryacquire}, \perp),\\ &ret(q, \texttt{tryacquire}, 0), inv(p, \texttt{release}, \perp), ret(p, \texttt{release}, \perp),\\ &inv(q, \texttt{tryacquire}, \perp), ret(q, \texttt{tryacquire}, 1)\rangle\end{aligned}$$

is the sequential history corresponding to the execution `acquire; tryaquire; release; tryaquire`. The histories of a concurrent implementation, however, may have overlapping operations and hence have the invocations and returns of operations separated, e.g., as in

$$\begin{aligned} h_c \mathrel{\widehat{=}} \langle & inv(p, \texttt{acquire}, \bot), inv(q, \texttt{tryacquire}, \bot), ret(p, \texttt{acquire}, \bot), \\ & inv(p, \texttt{release}, \bot), ret(p, \texttt{release}, \bot), ret(q, \texttt{tryacquire}, 0), \\ & inv(q, \texttt{tryacquire}, \bot), ret(q, \texttt{tryacquire}, 1)\rangle. \end{aligned}$$

However to be *legal*, a history should not have returns for which there has not been an invocation. This is captured in the following.

$$legal(h) \mathrel{\widehat{=}} \forall n : 1..\#h \bullet ret?(h(n)) \Rightarrow (\exists m : 1..\#h \bullet mp(h, m, n)).$$

The histories of abstract specifications are also *complete*, i.e., they have a return for each invocation. This is not necessarily the case for implementation histories. For example, the history $h_c {}^\frown \langle inv(q, \texttt{release}, \bot)\rangle$ is also legal although it is not complete. To make an implementation history complete, it is necessary to add additional returns for those operations which have been invoked and are deemed to have occurred, and to remove the remaining invocations without matching returns. We define a function *complete* to do the latter:

$$complete(h) \mathrel{\widehat{=}} \begin{cases} \langle\,\rangle & \text{if } h = \langle\,\rangle \\ complete(tail\ h) & \text{if } inv?(head\ h) \land NoRet(h) \\ \langle head\ h\rangle {}^\frown complete(tail\ h) & \text{otherwise} \end{cases}$$

where $NoRet(h) \mathrel{\widehat{=}} \forall n : 1..\#h \bullet \neg match(h, 1, n)$.

We define linearizability formally as follows. In this definition $Hist_R$ is the set of all histories that are sequences of returns, and $lin(h, hs)$ holds iff concurrent history h can be extended by adding such a sequence h_0 to form a legal history $h {}^\frown h_0$ such that *linrel* holds for $complete(h {}^\frown h_0)$ and hs. The relation $linrel(h, hs)$ holds if for some (total) bijective function f between indices of h and hs, f transforms h to hs (according to $maps(h, f, hs)$) and the order of non-overlapping operations is preserved (according to $order(h, f)$).

Definition 1 (*Linearizability*) A history $h : History$ is *linearizable* with respect to some sequential history hs iff $lin(h, hs)$ holds, where

$$lin(h, hs) \mathrel{\widehat{=}} \exists h_0 : \operatorname{seq} Hist_R \bullet legal(h {}^\frown h_0) \land linrel(complete(h {}^\frown h_0), hs)$$

and

$$\begin{aligned} maps(h, f, hs) &\mathrel{\widehat{=}} (\forall n : \operatorname{dom} f \bullet h(n) = hs(f(n))) \land \\ &\qquad (\forall m, n : \operatorname{dom} f \bullet mp(h, m, n) \Rightarrow f(n) = f(m) + 1) \\ order(h, f) &\mathrel{\widehat{=}} \forall m, n, m', n' : \operatorname{dom} f \bullet \\ &\qquad n < m' \land mp(h, m, n) \land mp(h, m', n') \Rightarrow f(n) < f(m') \\ linrel(h, hs) &\mathrel{\widehat{=}} \exists f : 1..\#h \rightarrowtail\!\!\!\!\twoheadrightarrow 1..\#hs \bullet maps(h, f, hs) \land order(h, f) \qquad \Box \end{aligned}$$

That is, history h of the concurrent implementation can be transformed into a sequential history hs such that the operations in hs do not overlap (each invocation is followed immediately by its matching return) and the order of non-overlapping operations in h is preserved in hs. For example, the histories h_s and h_c above are both complete and legal, and $linrel(h_c, h_s)$ holds, i.e., h_c is linearized by h_s.

Finally, we can lift the definition of linearizability of histories to specifications: a concrete specification is linearizable if all its histories are.

2.2 A Proof Method for Linearizability

The proof method for linearizability defined and applied in [8–10, 20] is based on showing that a concrete specification is a non-atomic refinement of the abstract one. The steps from [9] are summarised below.

2.2.1 Modelling the Algorithm in Z

The Z description of the implementation has one operation per line of pseudo-code, where each operation can be invoked by a given process. The concrete state consists of the shared memory, given as a global state GS and local state LS for each process. For spinlock, GS includes the value of the shared variable x (initially 1), and a variable $lock$ which has value $\{p\}$ when a process p currently has the global memory locked (and is $\varnothing$ otherwise).

$$\begin{array}{|l}
\hline GS \\
x : \{0, 1\} \\
lock : \mathbb{P}\,P \\
\hline
\#lock \leq 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline GSInit \\
GS \\
\hline
x = 1 \\
lock = \varnothing \\
\hline
\end{array}$$

For a given process, the local state LS is specified in terms of a program counter, $PC ::= 1|a1|\ldots|a8|t1|\ldots|t7|r1$, indicating which operation (i.e., line of code) will be performed next. The value 1 denotes that the process is not executing any of the three operations. The values ai, for $i \in 1\,..\,8$, denote the process is ready to perform the ith line of code of `acquire`, and similarly for ti and `tryacquire`. The value $r1$ denotes the process is ready to perform the first line of `release`.

$$\begin{array}{|l}
\hline LS \\
pc : PC \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline LSInit \\
LS \\
\hline
pc = 1 \\
\hline
\end{array}$$

In formalising lines of code in Z, we adopt the convention that the values that are not explicitly changed by an operation remain unchanged. For process p, we have an operation $A0_p$ corresponding to the invocation of the `acquire` operation, and an operation $A1_p$ corresponding to the line of code `while(1)`.

$$\begin{array}{|l}
\hline A0_p \\
\Xi GS \\
\Delta LS \\
\hline pc = 1 \land pc' = a1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline A1_p \\
\Xi GS \\
\Delta LS \\
\hline pc = a1 \land pc' = a2 \\
\hline
\end{array}$$

The operation $A2_p$ corresponds to the line of code `lock`. To model `if (x==1)`, we use two operations: $A3t_p$ for the case when $x = 1$, and $A3f_p$ for the case when $x = 0$.

$$\begin{array}{|l}
\hline A2_p \\
\Delta GS \\
\Delta LS \\
\hline pc = a2 \land lock = \varnothing \\
pc' = a3 \land lock' = \{p\} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline A3t_p \\
\Xi GS \\
\Delta LS \\
\hline pc = a3 \land x = 1 \\
pc' = a4 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline A3f_p \\
\Xi GS \\
\Delta LS \\
\hline pc = a3 \land x = 0 \\
pc' = a7 \\
\hline
\end{array}$$

The operations corresponding to the rest of `acquire` are modelled similarly. The two operations corresponding to `while(x==0)`, $A8t_p$ and $A8f_p$, are only enabled when the memory is not locked (and so x can be read from the global memory).

$$\begin{array}{|l}
\hline A4_p \\
\Delta GS \\
\Delta LS \\
\hline pc = a4 \\
x' = 0 \land pc' = a5 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline A5_p \\
\Delta GS \\
\Delta LS \\
\hline pc = a5 \\
pc' = a6 \land lock' = \varnothing \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline A6_p \\
\Xi GS \\
\Delta LS \\
\hline pc = a6 \\
pc' = 1 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline A7_p \\
\Delta GS \\
\Delta LS \\
\hline pc = a7 \\
pc' = a8 \land lock' = \varnothing \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline A8t_p \\
\Xi GS \\
\Xi LS \\
\hline pc = a8 \\
lock = \varnothing \land x = 0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline A8f_p \\
\Xi GS \\
\Delta LS \\
\hline pc = a8 \\
lock = \varnothing \land x = 1 \\
pc' = a1 \\
\hline
\end{array}$$

The operations for `tryacquire` are similar to those of `acquire`. Those for `release` are given below.

$$\begin{array}{|l}
\hline R0_p \\
\Xi GS \\
\Delta LS \\
\hline pc = 1 \land pc' = r1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline R1_p \\
\Delta GS \\
\Delta LS \\
\hline pc = r1 \land x' = 1 \land pc' = 1 \\
\hline
\end{array}$$

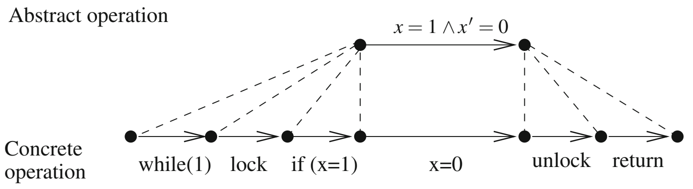


Fig. 2 Simulation of `acquire`

2.2.2 Proving Linearizability

Correctness requires showing all concrete histories are linearizable. Following [9], we use two proof steps for each operation of the concrete specification.
Step 1. Firstly, we need to show that the lines of code defining the concrete operations simulate the abstract operations. We identify one line of code as the *linearization step*, which must simulate the abstract operation, all others simulating an abstract skip. For example, for `acquire` we require that line `a4`, $x = 0$, simulates the abstract operation and all other lines simulate an abstract skip (see Fig. 2). To do this we define an abstraction relation relating the global (i.e., shared) concrete state space gs and abstract state space as. The abstraction relation $ABS(as, gs)$ for spinlock is simply $gs.x = as.x$.

We also need to define an invariant to enable the simulation of each line of code to be proven independently. In our example, to prove that $x = 0$ simulates the abstract operation, this invariant needs to ensure that at line `a4` we have $x = 1$. Such an invariant is stated in terms of the global and local concrete state spaces. Hence, the invariant $INV(gs, ls)$ must imply $ls.pc = a4 \Rightarrow x = 1$.

Each simulation is proved by one of five rules depending on whether the line of code is an invocation (beginning an operation), return (ending an operation) or internal step (neither an invocation nor return), and whether it occurs before or after the linearization step. A function $status(gs, ls)$ is defined to identify the linearization step. Before invocation, $status(gs, ls)$ is $IDLE$. After invocation but before the linearization step it is equal to $IN(in)$, where $in : In$ is the input to the abstract operation, and after the linearization step it is equal to $OUT(out)$, where $out : Out$ is the output of the abstract operation. For example, the simulation rule for an invocation is:

$$\begin{array}{l}
\forall as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet \\
\quad R(as, gs, ls) \wedge status(gs, ls) = IDLE \wedge COP(in, gs, ls, gs', ls') \Rightarrow \\
\qquad status(gs' ls') = IN(in) \wedge R(as, gs', ls') \\
\qquad \vee \\
\qquad (\exists as' : AS;\ out : Out \bullet \\
\qquad AOP(in, as, as', out) \wedge status(gs', ls') = OUT(out) \wedge R(as', gs', ls'))
\end{array}$$

where primed states, e.g., gs', represent post-states of operations whereas unprimed states, e.g., gs, represent pre-states; COP represents the meaning of a line of code from a concrete operation; and $R(as, gs, ls) = ABS(as, gs) \wedge INV(gs, ls)$. The disjunction in this rule allows the invocation to be either the linearization step or to simulate an abstract skip.

Step 2. Secondly, we need to prove non-interference between threads. This amounts to showing that a process p running the concrete code cannot, by changing the global concrete state space, invalidate the invariant which another process q relies on. For example, a process p should not be able to change the value of `x` when a process q is at line `a4` since this would invalidate the requirement on $INV(gs, ls)$ above. To do this we require a further invariant $D(ls, lsq)$ relating the local states of two process whose local states are ls and lsq. The non-interference rule is then:

$$\begin{aligned} &\forall as : AS; gs, gs' : GS; ls, ls', lsq : LS \bullet \\ &\quad ABS(as, gs) \wedge INV(gs, ls) \wedge INV(gs, lsq) \wedge D(ls, lsq) \wedge COP(gs, ls, gs', ls') \\ &\qquad \Rightarrow INV(gs', lsq) \wedge D(ls', lsq) \wedge status(gs', lsq) = status(gs, lsq). \end{aligned}$$

Of course, there is also an initialisation proof obligation:

$$\begin{aligned} &\forall gs : GSInit \bullet \exists as : ASInit \bullet \\ &\quad ABS(as, gs) \wedge (\forall ls : LSInit \bullet INV(gs, ls)) \wedge (\forall ls, lsq : LSInit \bullet D(ls, lsq)). \end{aligned}$$

As shown in [9] if these proof obligations are discharged then the concrete specification is linearizable with respect to the abstract. This is all well and good, however, so far this discussion has assumed a sequentially consistent memory model, and we now turn our attention to the weaker memory model TSO.

3 The TSO Memory Model

In the TSO architecture [23] each processor core uses a write buffer, which is a FIFO queue that stores pending writes to memory. A processor core performing a *write* to a memory location enqueues the write to the buffer and continues computation without waiting for the write to be committed to memory. Pending writes do not become visible to other cores until the buffer is *flushed*, which commits pending writes to memory. The value of a memory location *read* by a process is the most recent in that processor's local buffer, and only from the memory if there is no such value in the buffer. The use of local buffers allows a read by one process, occurring after a write by another, to return an older value as if it occurred before the write.

In general, flushes are controlled by the CPU, and from the programmer's perspective occur non-deterministically. However, a programmer may explicitly include a *fence* instruction in a program's code to force a flush to occur. Therefore, although TSO allows some non-sequentially consistent executions, it is used in many modern architectures on the basis that these can be prevented, where necessary, by programmers using fence instructions. In addition, a pair of *lock* and *unlock* commands allows a process to acquire sole access to the memory. Both commands include a fence which forces the store buffer of that process to be flushed completely.

So how does the TSO architecture affect the behaviour of the spinlock algorithm of Sect. 2? Since the `lock` and `unlock` commands include fences on TSO, writes to `x` by the `acquire` and `tryacquire` operations are not delayed. For efficiency, however, `release` does not have a fence and so its write to `x` can be delayed until a flush occurs. The spinlock implementation will still work correctly, the only effect that the absence of a fence has is that a subsequent `acquire` may be delayed until a `flush` occurs, or a `tryacquire` operation by a thread q may return 0 after the lock has been released by another thread p.

For example, if we use $(q, \texttt{tryacquire}(0))$ to denote process q performing a `tryacquire` operation and returning 0, and $\texttt{flush}(p)$ to denote the CPU flushing a value from process p's buffer, then the following execution is possible:

$$ex \mathrel{\widehat=} \langle (p, \texttt{acquire}), (p, \texttt{release}), (q, \texttt{tryacquire}(0)), \texttt{flush}(p) \rangle.$$

That is, the `tryacquire` returns 0 even though it occurs immediately after the `release`. This is because the $\texttt{flush}(p)$, which sets the value of x in memory to 1 has not yet occurred.

This can be considered correct behaviour since it is as if the `release` by process p occurred after, rather than before, the `tryacquire` of the process q, which is possible since the processes are independent. Although we want to accept this as a valid concurrent implementation, such a run is not linearizable using the definition given in Sect. 2. Therefore we adapt the definition of linearizability to work on the TSO model, paying particular attention to the role of flushes.

To model the above behaviour, the Z specification under TSO is modified from that of Sect. 2 as follows. The global state GS includes an additional variable modelling a buffer for each process. (Each buffer is a sequence of 0 and 1's.)

$$\begin{array}{|l} \multicolumn{1}{l}{GS} \\ \hline x : \{0,1\} \\ lock : \mathbb{P}P \\ buffer : P \to \operatorname{seq}\{0,1\} \\ \hline \#lock \leq 1 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{GSInit} \\ \hline GS \\ \hline x = 1 \\ lock = \varnothing \\ \forall p : P \bullet buffer(p) = \langle\,\rangle \\ \hline \end{array}$$

The local state schemas LS and $LSInit$, as well as the `acquire` operations $A0_p$, $A1_p$, $A3t_p$, $A3f_p$ and $A6_p$, from Sect. 2 are unchanged. The operation $A2_p$, corresponding to the `lock` at line `a2`, is only enabled when the buffer is empty, modelling the fact that the lock is a fence, i.e., a sequence of flush operations on p's buffer must occur immediately before $A2_p$ if the buffer is non-empty. The operation $A4_p$, corresponding to the line `x=0`, adds the value 0 to the buffer.

$$\begin{array}{|l}
\multicolumn{1}{l}{A2_p} \\ \hline
\Delta GS \\
\Delta LS \\ \hline
buffer(p) = \langle\,\rangle \\
pc = a2 \wedge lock = \varnothing \\
pc' = a3 \wedge lock' = \{p\} \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{A4_p} \\ \hline
\Delta GS \\
\Delta LS \\ \hline
pc = a4 \\
buffer'(p) = buffer(p) \frown \langle 0 \rangle \\
pc' = a5 \\ \hline
\end{array}$$

The operations $A5_p$ and $A7_p$ are only enabled when the buffer is empty, modelling that the buffer is completely flushed *before* unlocking the memory. We elide their definition.

The two operations corresponding to `while(x==0)`, $A80_p$ and $A81_p$, are only enabled when either x can be read from the buffer, i.e., $buffer \neq \langle\,\rangle$, or the buffer is empty and the memory is not locked (and so x can be read from the global memory).

$$\begin{array}{|l}
\multicolumn{1}{l}{A8t_p} \\ \hline
\Xi GS \\
\Xi LS \\ \hline
pc = a8 \\
buffer(p) = \langle\,\rangle \Rightarrow lock = \varnothing \wedge x = 0 \\
buffer(p) \neq \langle\,\rangle \Rightarrow last\,buffer(p) = 0 \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{A8f_p} \\ \hline
\Xi GS \\
\Delta LS \\ \hline
pc = a8 \\
buffer(p) = \langle\,\rangle \Rightarrow lock = \varnothing \wedge x = 1 \\
buffer(p) \neq \langle\,\rangle \Rightarrow last\,buffer(p) = 1 \\
pc' = a1 \\ \hline
\end{array}$$

The operations for `tryacquire` and `release` are similarly modified. We also have an operation, $Flush_{cpu}$ (where $cpu \in P$ is a special value denoting the CPU), corresponding to a CPU-controlled flush which outputs the process whose buffer it flushes. This operation must repeatedly occur to empty the buffer before operations $A3t_p$, $A3f_p$, $A5_p$ and $A7_p$ can occur.

$$\begin{array}{|l}
\multicolumn{1}{l}{Flush_{cpu}} \\ \hline
\Delta GS \\
p! : P \\ \hline
lock = \varnothing \vee lock = \{p!\} \\
buffer(p!) \neq \langle\,\rangle \Rightarrow x' = head\,buffer(p!) \wedge buffer'(p!) = tail\,buffer(p!) \\
buffer(p!) = \langle\,\rangle \Rightarrow x' = x \wedge buffer'(p!) = buffer(p!) \\ \hline
\end{array}$$

In our approach to modelling algorithms on TSO, we assume that a flush is only executed by the CPU process, and that this process is different from all other processes. We also assume that, in a history of the specification, invocations of flushes are immediately followed by their returns.

3.1 TSO-Linearizability

Having seen how TSO affects the behaviour of the concurrent objects, we need to furnish the model with an appropriate correctness condition. Although most of the work on linearizability has assumed an SC architecture, some work has been undertaken for TSO (e.g., see [4, 11, 15, 24]). In particular, a definition for linearizability on TSO has been proposed in [11], where the role of local buffers and flushes is taken into account in the following way: since the flush of a process's buffer is sometimes the point that the effect of an operation becomes globally visible, the flush can be viewed as being the return of the operation. For example, the flush of a variable, such as `x`, after an operation, such as `release`, can be taken as the return of that operation. Using this idea, the `release` operation begins with its invocation but returns with the flush which writes its change to `x` to the global memory. Thus the return point of an operation on a TSO architecture is not necessarily the point where the operation ceases execution, but can be any point up to the last flush of the variables written by that operation.

So we will define TSO-linearizability in terms of history transformations, where the actual return of an operation may be moved to the return of a corresponding flush [11]. We wish to reuse as much of the standard definition of linearizability as possible, so we first transform a concurrent history (which includes flush events) to a history in which the return of the `release` is moved to a corresponding flush, and events corresponding to flushes are removed. This produces a new history with potentially more overlapping operation calls than the original. The original concurrent history is judged to be TSO-linearizable iff the new transformed history is linearizable.

Calculating the return of an operation. To define the history transformation, we need to calculate the position of the flush corresponding to an operation's return. This is done by a function *mpf* (standing for *matching pair flush*) which in turn uses *mp* defined in Sect. 2.1. A flush acts as a return for an operation, i.e., makes its effects visible globally, when it writes the last variable which was updated by that operation to memory.

We extend the definition of *Event* to include a natural number representing the size of the buffer of the process performing the event. This number is always zero in the case of the *cpu* process. A *TSO history* is a sequence of such events.

$$\begin{aligned} &Event_{TSO} \mathrel{\widehat{=}} inv\langle\langle I \times P \times In \times \mathbb{N}\rangle\rangle \mid ret\langle\langle I \times P \times Out \times \mathbb{N}\rangle\rangle \\ &History_{TSO} \mathrel{\widehat{=}} \mathrm{seq}\ Event_{TSO} \end{aligned}$$

Let $h(m).bs$ denote the size of the buffer of process $h(m).\pi$ at point m in the history h. Consider an operation of a history h whose invocation is at point m and whose return is at point n. If the buffer is empty when the operation is invoked, then the number of flushes to be performed before the operation returns is equal to the size of the buffer at the end of the operation, i.e., $h(n).bs$; if this number is 0 then the return does not move. Similarly, if an operation contains a fence then the number of flushes before the operation returns is also equal to $h(n).bs$. In all other cases, we need to determine whether the operation has written to any global variables. If it has written to one or more global variables then again the number of flushes to be performed before the operation returns is $h(n).bs$.

To determine whether an operation has written to global variables, we compare the size of the buffer at the start and end of the operation taking into account any *modifying flushes*, i.e., flushes performed when the buffer is not empty, in between. Let $nf(h, p, m, n)$ denote the number of modifying flushes of process p's buffer from point m up to and including point n in h. The number of writes between the two points is given by $nw(h, p, m, n) \mathrel{\widehat{=}} h(n).bs - h(m).bs + nf(h, p, m, n)$.

The predicate *mpf* is then defined below where m, n and l are indices in h such that (m, n) is a matching pair of an operation and l corresponds to the point to which the return of the matching pair must be moved.

$$\begin{aligned} mpf(h, m, n, l) \mathrel{\widehat{=}} {} & \exists p : P \bullet h(m).\pi = p \wedge mp(h, m, n) \wedge n \leq l \wedge h(m).i \neq \texttt{flush} \wedge \\ & (\textsf{if } nw(h, p, m, n) = 0 \vee h(n).bs = 0 \textsf{ then } l = n \\ & \textsf{else } h(l) = ret(cpu, \texttt{flush}, p, 0) \wedge nf(h, p, n, l) = h(n).bs) \end{aligned}$$

The first line of the if states that $l = n$ if no items are put on the buffer by the operation invoked at point m, or all items put on the buffer have already been flushed when the operation returns. The second line states that l corresponds to a flush of p's buffer and the number of flushes between n and l is precisely the number required to flush the contents of the buffer at n.

Example 1 Consider the following concurrent history (recall the final element of each event is the relevant process's buffer size):

$$\begin{aligned} he \mathrel{\widehat{=}} {} & \langle inv(p, \texttt{acquire}, \perp, 0), inv(q, \texttt{tryacquire}, \perp, 0), ret(p, \texttt{acquire}, \perp, 0), \\ & inv(p, \texttt{release}, \perp, 0), ret(p, \texttt{release}, \perp, 1), ret(q, \texttt{tryacquire}, 0, 0), \\ & inv(cpu, \texttt{flush}, \perp, 0), ret(cpu, \texttt{flush}, p, 0)\rangle \end{aligned}$$

For the `acquire` operation we get $mpf(he, 1, 3, 3)$, for the `tryacquire` operation we get $mpf(he, 2, 6, 6)$, and for the `release` operation we get $mpf(he, 4, 5, 8)$. That is, the matching flush for the `release` operation is the final one in the history above, but the other operations return when they complete. □

Defining a history transformation. We define our transformation *Trans* which moves the return of each operation, when necessary, to the flush which makes its global behaviour visible to other processes. The transformation also removes all flushes and results in a history of type *History* (rather than $History_{TSO}$). Therefore, the types of events in a concrete history h and its transformed history $Trans(h)$ will be different; we use $SC(inv(p, i, v, n)) = inv(p, i, v)$ and $SC(ret(p, i, v, n)) = ret(p, i, v)$ to convert an event of type $Event_{TSO}$ to type *Event*.

The formal definition of *Trans* is based on identifying the matching pairs, and ordering them by the positions that invocations and returns are moved to. The key point is that the positions that returns get moved to are different for each event, so we can order them, and this defines our new history.

Definition 2 (*Trans*) Let h be a history of the concrete specification. We define a set $S(h) \widehat{=} \{(m, n, l, x) | mpf(h, m, n, l) \wedge x \in \{m, l\}\}$, which has one tuple (m, n, l, x) for each event e in the transformed history. The first three elements of each tuple correspond to a matching pair (m, n) of a non-flush operation op in the original history and the point l to which its return is moved. The event e will be either the invocation or return of op. The final element x denotes the position in the original history of the event corresponding to e. If $x = m$ then the event in the original history is an invocation and e is an invocation. If $x = l$ then the event in the original history is a return (possibly of a flush) and e is a return.

We can order the elements of $S(h)$ by their 4th elements: $x_1 < x_2 < \cdots < x_k$ where $k = \#S(h)$. Then $Trans(h)$ is a history with length k defined (for $i : 1..k$) as:

$$Trans(h)(i) = \begin{cases} SC(h(x_i)), & \text{if } (x_i, n, l, x_i) \in S(h), \text{ for some } n \text{ and } l \\ SC(h(n)), & \text{if } (m, n, x_i, x_i) \in S(h), \text{ for some } m \end{cases}$$ □

Example 2 Consider the history he in Example 1. The elements of set $S(he)$ are ordered as follows: $(x_1, 3, 3, x_1)$, $(x_2, 6, 6, x_2)$, $(1, 3, x_3, x_3)$, $(x_4, 5, 8, x_4)$, $(2, 6, x_5, x_5)$, $(4, 5, x_6, x_6)$ (where $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 6$ and $x_6 = 8$). Thus, $Trans(he)(1) = he(1)$ since $x_1 = 1$, and $Trans(he)(6) = he(8)$ since $x_6 = 8$. Overall $Trans(he)$ is as follows where the return of the `release` has been moved as required:

$$\langle inv(p, \texttt{acquire}, \bot), inv(q, \texttt{tryacquire}, \bot), ret(p, \texttt{acquire}, \bot),$$
$$inv(p, \texttt{release}, \bot), ret(q, \texttt{tryacquire}, 0), ret(p, \texttt{release}, \bot)\rangle$$ □

A key part of adapting the definition of linearizability from Sect. 2 to TSO is formalising what we mean by a matching pair of invocations and returns. The formal definition of the function mp requires that for all k between m and n, $h(k)$ is not an invocation or return event of p. This is not true for our transformed histories on TSO since operations by the same process may now overlap. So, we calculate the matching pairs for a transformed history from those of the original history. This is done by subtracting from the positions m and n of the matching pairs the number of flushes that have occurred before them. The matching pairs of a transformed history $Trans(h)$ are given by $mp_{TSO}(h, m_1, n_1)$ defined over the original history h as follows.

$$mp_{TSO}(h, m_1, n_1) \widehat{=} \exists m, n, l \bullet mpf(h, m, n, l) \wedge$$
$$m_1 = m - \sum_{p:P} nf(h, p, 1, m) \wedge n_1 = l - \sum_{p:P} nf(h, p, 1, l)$$

Example 3 For the transformed history $Trans(he)$ in Example 2, the matching pairs are $mp_{TSO}(he, 1, 3)$, $mp_{TSO}(he, 2, 5)$ and $mp_{TSO}(he, 4, 6)$. □

A TSO history is legal if (i) it does not have returns for which there has not been an invocation (as for standard histories), and (ii) the number of modifying flushes performed on a process's buffer never exceeds the number of values placed in the buffer.

$$\begin{aligned} legal_{TSO}(h) \mathrel{\widehat{=}}\ & legal(h) \land \\ & (\forall p : P; n : 1..\#h \bullet h(n).\pi \neq p \Rightarrow nf(h, p, 1, n) = 0) \land \\ & (\forall n, m' : 1..\#h \bullet ret?(h(n)) \land (\forall n < k < m' \bullet h(k).\pi \neq h(n).\pi) \\ & \quad \Rightarrow nf(h, h(n).\pi, n, m') \leq h(n).bs) \end{aligned}$$

We adopt the definition of TSO-linearizability from [11]. After extending an incomplete concrete history with flush operations (to empty all buffers) and returns, we apply *Trans* to it before matching it to an abstract history. Let $Hist_{FR}$ be the set of histories that are sequences of complete flush operations and returns.

Definition 3 (*TSO-linearizability*) A history $h : History$ is *TSO-linearizable* with respect to some sequential history hs iff $lin_{TSO}(h, hs)$ holds, where

$$\begin{aligned} lin_{TSO}(h, hs) \mathrel{\widehat{=}} \exists h_0 : Hist_{FR} \bullet & legal_{TSO}(h^\frown h_0) \land \\ & linrel_{TSO}(Trans(complete(h^\frown h_0)), hs, h^\frown h_0) \end{aligned}$$

where

$$\begin{aligned} maps_{TSO}(h', f, hs, h) \mathrel{\widehat{=}}\ & (\forall n : \mathrm{dom} f \bullet h'(n) = hs(f(n))) \land \\ & (\forall m, n : \mathrm{dom} f \bullet mp_{TSO}(h, m, n) \Rightarrow f(n) = f(m) + 1) \\ order_{TSO}(h, f) \mathrel{\widehat{=}}\ & \forall m, n, m', n' : \mathrm{dom} f \bullet \\ & mp_{TSO}(h, m, n) \land mp_{TSO}(h, m', n') \land n < m' \Rightarrow f(n) < f(m') \\ linrel_{TSO}(h', hs, h) \mathrel{\widehat{=}}\ & \exists f : 1..\#h' \rightarrowtail 1..\#hs \bullet \\ & maps_{TSO}(h', f, hs, h) \land order_{TSO}(h, f) \end{aligned}$$

□

As before we lift TSO-linearizability to the level of specifications in the same manner: a concrete specification is TSO-linearizable if all its histories are TSO-linearizable.

4 Using a Coarse-Grained Abstraction

Any proof method for proving TSO-linearizability will be complicated by having to deal with both the inherent interleaving handled by linearizability and the additional potential overlapping of concrete operations resulting from moving operation returns to associated flushes. For example, in spinlock, a process may perform a `release` but not have its buffer flushed before invoking its next operation.

To handle this complexity, we use an intermediate specification, between the abstract and concrete, to split the original proof obligations into two simpler components. The first, between the concrete and intermediate specifications, deals with the underlying linearizability, and the second, between intermediate and abstract, deals

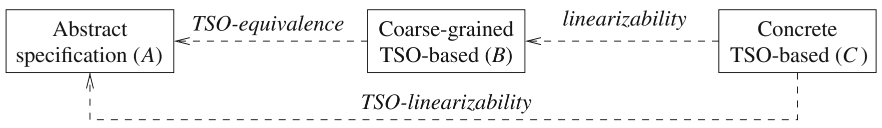


Fig. 3 Verification chain

with the effects of local buffers. The intermediate specification is a *coarse-grained abstraction* that captures the semantics of the concrete specification with no fine-grained interleaving of operations by different processes. We describe how to define such a coarse-grained abstraction in the next subsection.

An overview of the approach is provided in Fig. 3. The concrete specification is proved linearizable, using the existing proof method, with respect to a coarse-grained abstraction which, ignoring flushes, has the same granularity of operations as the abstract specification. Hence, TSO effects do not complicate the linearizability proof. These are instead dealt with when we show that the coarse-grained abstraction is *TSO-equivalent* to the abstract specification as detailed in this section. It can be proved that these two steps imply that the concrete specification is TSO-linearizable with respect to the abstract specification.

4.1 Defining the Coarse-Grained Abstraction

The coarse-grained abstraction is constructed by adding local buffers to the abstract specification. Thus, it is still a description on the TSO architecture – since it has buffers and flushes – but does not decompose the operations. The state space is the abstract state space with the addition of a buffer for each process (as in the concrete state space GS). Like in the concrete state space, all buffers are initially empty. Hence for spinlock we have:

BS

$x : \{0,1\}$
$\mathit{buffer} : P \to \mathrm{seq}\{0,1\}$

BSInit

BS

$x = 1 \land \forall p : P \bullet \mathit{buffer}(p) = \langle\rangle$

Each operation is like that of the abstract specification except that

- a read is replaced by a read from the process's buffer or from memory, i.e., the operation refers to the latest value of the variable in the buffer, if there is one, and to the value in memory otherwise,
- a write is replaced by a write to the buffer (unless the corresponding concrete operation has a fence),
- because we have buffers in the intermediate state space we need to include fences and flushes: the buffer is set to empty when the corresponding concrete operation has a fence, and a flush is modelled as a separate operation.

For example, for the abstract operation $Acquire_p$, $x = 1$ represents a read, and $x' = 0$ represents a write. Using the above heuristic, we replace $x = 1$ by $(buffer(p) \neq \langle\,\rangle \Rightarrow \text{last}\, buffer(p) = 1) \wedge (buffer(p) = \langle\,\rangle \Rightarrow x = 1)$ since the latest value of x is that in the buffer when the buffer is not empty, and the actual value of x otherwise. We also replace $x' = 0$ by $buffer'(p) = \langle\,\rangle \wedge x' = 0$ since the corresponding concrete operation has a fence. Similarly, while the operation $TryAcquire_p$ writes directly to x and sets the buffer to empty (since it has a fence), the operation $Release_p$ writes only to the buffer.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Acquire_p} \\
\Delta BS \\
\hline
buffer(p) \neq \langle\,\rangle \Rightarrow last\, buffer(p) = 1 \\
buffer(p) = \langle\,\rangle \Rightarrow x = 1 \\
buffer'(p) = \langle\,\rangle \wedge x' = 0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Release_p} \\
\Delta BS \\
\hline
buffer'(p) = buffer(p) \frown \langle 1 \rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{TryAcquire_p} \\
\Delta BS \\
out! : \{0, 1\} \\
\hline
\textbf{if } (buffer(p) \neq \langle\,\rangle \wedge last\, buffer(p) = 1) \vee (buffer(p) = \langle\,\rangle \wedge x = 1) \\
\textbf{then } buffer'(p) = \langle\,\rangle \wedge x' = 0 \wedge out! = 1 \\
\textbf{else } buffer'(p) = \langle\,\rangle \wedge x' = 0 \wedge out! = 0 \\
\hline
\end{array}
$$

Note that $x' = 0$ holds in the else-predicate of $TryAcquire_P$ since if the buffer is empty, x is 0 and does not change, and if the buffer is not empty, the last element in buffer is 0 and the buffer is completely flushed by the `lock` command in `tryacquire`.

Finally, the course-grained abstraction is completed with the $Flush_{cpu}$ operation. As in the concrete specification, this operation is performed by the CPU process.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Flush_{cpu}} \\
\Delta BS \\
p! : P \\
\hline
buffer(p!) \neq \langle\,\rangle \Rightarrow x' = head\, buffer(p!) \wedge buffer'(p!) = tail\, buffer(p!) \\
buffer(p!) = \langle\,\rangle \Rightarrow x' = x \wedge buffer'(p!) = buffer(p!) \\
\hline
\end{array}
$$

The coarse-grained abstraction is chosen purposefully to reflect the abstract specification; this facilitates the final part of the proof. The inclusion of buffers and flush operations, however, means it can be shown to linearize the concrete specification using standard proof methods.

Example 4 The concrete history *he* of Example 1 (with the buffer sizes removed from the events) is complete and legal, and linearized by the intermediate history

$$\begin{aligned} hs \mathrel{\widehat{=}} \langle & inv(p, \texttt{acquire}, \bot), ret(p, \texttt{acquire}, \bot), inv(p, \texttt{release}, \bot), \\ & ret(p, \texttt{release}, \bot), inv(q, \texttt{tryacquire}, \bot), ret(q, \texttt{tryacquire}, 0), \\ & inv(cpu, \texttt{flush}, \bot), ret(cpu, \texttt{flush}, p)\rangle \end{aligned} \qquad \square$$

Correctness requires showing all concrete histories are linearizable. The key point for us is that, for this portion of the correctness proof, we do not have to adapt the existing proof method.

4.2 From Coarse-Grained to Abstract Specification

Overall, we want to show the correctness of the concrete specification with respect to the abstract one. The previous section has defined an intermediate, coarse-grained abstraction, and the inclusion of local buffers in this intermediate specification avoided us needing to deal with the effects of the TSO architecture. In this subsection we introduce the idea of *TSO-equivalence* which allows us to move between intermediate and abstract specification via a history transformation which we define below. Correctness involves showing every history of the intermediate specification is transformed to a history of the abstract one.

The histories of the intermediate specification are sequential, i.e., returns of operations occur immediately after their invocations, but the specification includes buffers and flush operations. The transformation we define now turns the TSO histories of the intermediate specification into histories of an abstract one, i.e., without buffers, with the same behaviour. It does this according to the principle adopted in Sect. 3.1, i.e., it moves the return of an operation to the flush that makes its global behaviour visible. To keep histories sequential, we also move the invocation of the operation to immediately before the return.

The history transformation *TRANS* relies on the fact that the intermediate histories are sequential, i.e., comprise a sequence of matching pairs. Each matching pair of a history is either moved to the position of the flush which acts as its return (given by *mpf*), or left in the same position relative to the other matching pairs. The transformation also removes all flushes from the history. In a manner similar to *Trans* of Sect. 3.1, this is formalised in the following definition.

Definition 4 (*TRANS*) Let hs be a history of the intermediate specification. Let $T(hs) = \{(m, n, l) | mpf(hs, m, n, l)\}$, and $k = \#T(hs)$. We can order elements of $T(hs)$ by the 3rd element in the tuple: $l_1 < l_2 < \cdots < l_k$. Then $TRANS(hs)$ is an abstract history with length $2k$ defined (for $i : 1..2k$) as:

$$TRANS(hs)(i) = \begin{cases} SC(hs(n)) & \text{if } i \text{ is even and } (m, n, l_{i/2}) \in T(hs) \\ SC(hs(m)) & \text{if } i \text{ is odd and } (m, n, l_{(i+1)/2}) \in T(hs) \end{cases}$$

$\square$

The definition assigns each odd position in $\mathit{TRANS}(\mathit{hs})$ to the invocation of an event in hs and the immediately following even position to that event's return. The order of the invocations/return pairs corresponds to the order of the points to which their returns are moved according to mpf.

Example 5 Given the intermediate-level history hs in Example 4, the indices which are related by mpf are as follows: for the `acquire` operation we get $\mathit{mpf}(\mathit{hs}, 1, 2, 2)$, for the `release` operation we get $\mathit{mpf}(\mathit{hs}, 3, 4, 8)$, and for the `tryacquire` operation we get $\mathit{mpf}(\mathit{hs}, 5, 6, 6)$. The tuples in $T(\mathit{hs})$ are then ordered: $(1, 2, l_1)$, $(5, 6, l_2)$, $(3, 4, l_3)$ (where $l_1 = 2$, $l_2 = 6$ and $l_3 = 8$). Thus, $\mathit{TRANS}(\mathit{hs})(1) = \mathit{hs}(1)$ since 1 is odd and $(1, 2, l_1) \in T(\mathit{hs})$, whereas, $\mathit{TRANS}(\mathit{hs})(6) = \mathit{hs}(4)$ as 6 is even and $(3, 4, l_3) \in T(\mathit{hs})$. Overall, $\mathit{TRANS}(\mathit{hs})$ is the following:

$$\langle \mathit{inv}(p, \texttt{acquire}, \bot), \mathit{ret}(p, \texttt{acquire}, \bot), \mathit{inv}(q, \texttt{tryacquire}, \bot),$$
$$\mathit{ret}(q, \texttt{tryacquire}, 0), \mathit{inv}(p, \texttt{release}, \bot), \mathit{ret}(p, \texttt{release}, \bot)\rangle\,. \qquad \square$$

Finally, we can define TSO-equivalence:

Definition 5 (*TSO-equivalence*) An intermediate specification B is *TSO-equivalent* to an abstract specification A if for every legal history hs of B, $\mathit{TRANS}(\mathit{hs})$ is a history of A. □

It is now possible to show that the method of proving TSO-linearizability using coarse-grained abstractions is sound (a proof is provided in [12]).

Theorem 1 *If C is linearizable with respect to B and B is TSO-equivalent to A, then C is TSO-linearizable with respect to A.* □

5 Case Study: Work-Stealing Deque

The spinlock example is fairly simple: Firstly, it has only one global variable and hence all values stored in the write buffers are values of that variable. Generally, there would be more than one global variable and hence the buffer values need to be annotated in some way to identify the associated global variable.

Secondly, the global variables in the abstract specification are identical to those of the concrete specification. The consequence of this is that when the coarse-grained abstraction is derived from the abstract specification, it has the same buffer values as the concrete specification. This is actually required for the approach presented in Sect. 4, but such a relationship between abstract and concrete global variables does not always hold.

We now show how our approach can be adapted to handle more general algorithms. In particular, we extend the approach to include an extra specification, between the abstract specification and coarse-grained abstraction, which is buffer-free (like the

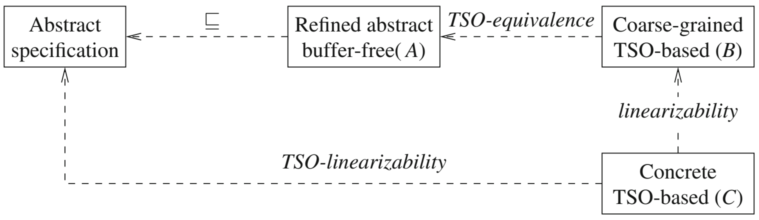


Fig. 4 Verification chain

abstract specification) but has the same state representation as the concrete specification. An overview of the approach appears in Fig. 4.

As an example we specify the Chase-Lev work-stealing deque [6]. Work-stealing deques (double-ended queues) are often used for load balancing in multiprocessor systems. Each worker process has a deque, which it uses to record tasks to be performed. Thus, a worker executes `put` and `take` operations that, respectively, add tasks to and remove tasks from its deque. Load balancing is achieved by allowing other, so-called "thief" processes, whose own deques are empty, to execute `steal` operations that remove elements from the deque. To avoid contention between the worker and thief processes, `put` and `take` operate at the opposite end of the deque from `steal` operations — a worker adds and removes tasks at the tail, whereas thieves steal tasks from the head. Contention between the worker and thieves, therefore, only occurs when the deque has one element. The Chase-Lev work-stealing deque, though linearizable on a sequentially consistent architecture, is not linearizable on TSO without the introduction of fences [17]. Here, we show that one of the fences required to preserve linearizability [17] can be removed, provided the correctness condition is weakened to TSO linearizability.

5.1 Abstract Specification

Our abstract specification assumes that the deque holds a maximum of W tasks. At the abstract level, we leave the behaviour undefined when more than W tasks are added to the deque. The state is specified in terms of a sequence of tasks of type *Task* which is initially empty.

```
─AS──────────────────────
│ tasks : seq Task
├──────────
│ #tasks ≤ W
└────────────────────────

─ASInit──────────────────
│ AS
├──────────
│ tasks = ⟨ ⟩
└────────────────────────
```

The operations Put_p and $Take_p$ model the worker p's operations on the deque; adding and removing tasks at the tail of the deque. When the deque is empty, $Take_p$

will return a special value *empty*. The operation $Steal_q$ models a thief process q (where $q \neq p$) removing a task from the head of the deque, when it is not empty.

$$\begin{array}{|l}
\underline{\textit{Put}_p\qquad\qquad} \\
\Delta AS \\
task? : Task \\
\hline
\#tasks < W \Rightarrow \\
\quad tasks' = tasks \frown \langle task? \rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\underline{\textit{Take}_p\qquad\qquad} \\
\Delta AS \\
task! : Task \cup \{empty\} \\
\hline
tasks = \langle\, \rangle \Rightarrow task! = empty \\
tasks \neq \langle\, \rangle \Rightarrow task = task' \frown \langle task! \rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\underline{\textit{Steal}_p\qquad\qquad\qquad\qquad\qquad\qquad} \\
\Delta AS \\
task! : Task \cup \{empty\} \\
\hline
tasks = \langle\, \rangle \Rightarrow task! = empty \\
tasks \neq \langle\, \rangle \Rightarrow tasks = \langle task! \rangle \frown tasks' \\
\hline
\end{array}$$

5.2 Concrete Specification

An implementation of the Chase-Lev work-stealing deque (taken from [18]) is given in Fig. 5. It comprises a cyclic array of `W` tasks and two pointers: `H`, the head pointer, points to the oldest task in the deque, and `T`, the tail pointer, to the first unused position in the array. When `T=H`, the deque is empty. The pointers are non-wrapping, i.e., if a pointer has the value `i` it points to the array element at position `i mod W`.

There are three operations: `put` enqueues a task to the tail of the deque, `take` dequeues from its tail, and `steal` dequeues from its head. `put` and `take` are performed by a worker process, and `steal` by a thief process which removes tasks from the worker's deque in order to balance the workload in the system. `take` and `steal` return `EMPTY` when applied to an empty deque. To ensure correct behaviour on TSO, a single fence has been added at line `t3` in $Take_p$.

There are three extra variables: `h` and `t` denoting local copies of the pointer values `H` and `T`, respectively, and `task` denoting a local copy of a task.

The interesting behaviour is in the way that the `take` and `steal` operations interact when called concurrently. To take the task at position `t=T-1`, the worker process decrements `T` to equal `t` (line `t2`) thereby publishing its intent to take that task. This publication, ensured by the fence at line `t3`, means subsequent thieves will not try to steal the task at position `t`. It then reads `H` into `h` and if `t > h` knows that there is more than one task in the deque and it is safe to take the task at position `t`, i.e., no thief process can concurrently steal it.

If `t < h` the worker knows the deque is empty and sets `T` to equal `h`. The final possibility is that `t=h`. In this case, there is one task on the deque and conflict with a thief may arise. To deal with this conflict, both the `take` and `steal` operations

```
int H = 0, T = 0;
Task [] tasks = new Task[W];

int h, t;
Task task;

put(Task task) {
  p1  t = T;
  p2  tasks[t mod W] = task;
  p3  T = t + 1;
}


steal() {
  s1 while (true) {
  s2     h = H;
  s3     t = T;
  s4     if (h >= t)
  s5         return EMPTY;
  s6     task = tasks[h mod W];
  s7     if (!CAS(H, h, h+1))
             // goto line s1
  s8         continue;
  s9     return task;
     }
}
```

```
take() {
  t1  t = T - 1;
  t2  T = t;
  t3  fence();
  t4  h = H;
  t5  if (t > h)
  t6      return tasks[t mod W];
  t7  if (t < h) {
  t8      T = h;
  t9      return EMPTY;
      }
      // t = h
  t10 T = h+1;
  t11 if (!CAS(H, h, h+1))
  t12     return EMPTY;
  t13 return tasks[t mod W];
}
```

Fig. 5 Chase-Lev algorithm

employ an atomic CAS (compare-and-swap) operation. An operation `CAS(x,y,z)` checks whether `x` equals `y` and, if so, updates `x` to `z` and returns true, otherwise it returns false leaving `x` unchanged. The CAS is atomic, and the update is immediately written to memory since the CAS operation also implements a fence.

The `steal` operation reads the deque's head and tail into `h` and `t`, and if the deque is not empty tries to increment `H` from `h` to `h+1` using the CAS at line `s7`. If it succeeds, the value of `H` has not been changed since read into the local variable `h` and hence the thief has stolen the task. The `take` operation works similarly. If `t=h`, rather than decrementing `T` to take the task, the worker increments `H`. Therefore, after decrementing `T`, if the worker finds `t=h`, it restores `T` to its original value (line `t10`) and then tries to increment `H` from `h` to `h+1` using the CAS at line `t11`.

To specify the Chase-Lev deque implementation in Z, we need to allow values of more than one variable in the write buffers. The buffer can contain array entries, i.e., tasks, as well as natural numbers corresponding to the global variable `T`. The value of `H` is never in the buffer since it is only changed in the CAS operations and hence written directly to memory.

A general way to model such a buffer is illustrated in [22]. Let *Id* be a set of identifiers, one for each global variable whose value may be in the buffer, and let *U* be the union of the types of all such global variables. Then *buffer* is of type $P \rightarrow \mathrm{seq}(Id \times U)$ where *P* is the set of all processes. For example, the Chase-Lev deque has `W+1` global variables whose value may be in the buffer, each of the `W` array entries and the pointer `T`. Hence, we define $Id == 0..W$ such that the values $0..W-1$ identify array entries in the buffer and W identifies a value of T. Representing the array by a sequence of *Tasks* we have

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{GS}}} \\
tasks : \operatorname{seq} Task \\
H, T : \mathbb{N} \\
buffer : P \rightarrow \operatorname{seq}(Id \times (Task \cup \mathbb{N})) \\
\hline
\#tasks = W \\
\forall p : P \bullet \forall i : \operatorname{dom} buffer(p) \bullet \\
\quad first(buffer(p)(i)) = W \\
\quad \Leftrightarrow \\
\quad second(buffer(p)(i)) \in \mathbb{N} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{GSInit}}} \\
GS \\
\hline
H = T = 0 \\
\forall p : P \bullet buffer(p) = \langle\,\rangle \\
\hline
\end{array}$$

The local state is defined as follows, where $PC ::= 1 | p1 | \ldots | p3 | t1 | \ldots | t13 | s1 | \ldots | s9$.

$$LS \mathrel{\widehat{=}} [h, t : \mathbb{N}; task : Task; pc : PC] \qquad LSInit \mathrel{\widehat{=}} [LS | pc = 1]$$

To simplify the specification of the operations, we write $\overline{x}(p)$ to denote the value of x read by a process p. This value is either the most recent in its buffer or, when no such value exists, the value of the global variable x.

The lines of code of the operation `put` are modelled as follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{Put0_p}} \\
\Xi GS; \Delta LS \\
task? : Task \\
\hline
pc = 1 \wedge pc' = p1 \\
task' = task? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\underline{Put1_p}} \\
\Xi GS; \Delta LS \\
\hline
pc = p1 \wedge pc' = p2 \\
t' = \overline{T}(p) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{Put2_p}} \\
\Delta GS; \Delta LS \\
\hline
pc = p2 \wedge pc' = p3 \\
buffer'(p) = \\
\quad buffer(p) \frown \langle (t \bmod W, task) \rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\underline{Put3_p}} \\
\Delta GS; \Delta LS \\
\hline
pc = p3 \\
buffer'(p) = buffer(p) \frown \langle (W, t+1) \rangle \\
pc' = 1 \\
\hline
\end{array}$$

The first 6 lines of the operation `take`, corresponding to a task being returned without conflict, are specified as follows. Note that 1 is added to the index of the output task, in $Take6_p$, to convert the array index (between 0 and $W-1$) to an index of the sequence *tasks* (between 1 and W).

$$\begin{array}{|l}\hline \mathit{Take}0_p \\ \Xi GS;\ \Delta LS \\ \hline pc = 1 \land pc' = t1 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}1_p \\ \Xi GS;\ \Delta LS \\ \hline pc = t1 \land pc' = t2 \\ t' = \overline{T}(p) - 1 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}2_p \\ \Delta GS;\ \Delta LS \\ \hline pc = t2 \land pc' = t3 \\ \mathit{buffer}'(p) = \mathit{buffer}(p) \frown \langle (W, t) \rangle \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}3_p \\ \Delta GS;\ \Delta LS \\ \hline \mathit{buffer}(p) = \langle\,\rangle \\ pc = t3 \land pc' = t4 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}4_p \\ \Xi GS;\ \Delta LS \\ \hline pc = t4 \\ h' = H \\ pc' = t5 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}5t_p \\ \Xi GS;\ \Delta LS \\ \hline pc = t5 \\ t > h \\ pc' = t6 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}5f_p \\ \Xi GS;\ \Delta LS \\ \hline pc = t5 \\ t \leq h \\ pc' = t7 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}6_p \\ \Delta GS;\ \Delta LS \\ \mathit{task}! : \mathit{Task} \\ \hline pc = t6 \land \mathit{task}! = \overline{\mathit{tasks}(t \bmod W + 1)}(p) \land pc' = 1 \\ \hline \end{array}$$

The next 3 lines of code correspond to *empty* being returned:

$$\begin{array}{|l}\hline \mathit{Take}7t_p \\ \Xi GS;\ \Delta LS \\ \hline pc = t7 \land pc' = t8 \\ t < h \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}7f_p \\ \Xi GS;\ \Delta LS \\ \hline pc = t7 \land pc' = t10 \\ t \geq h \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}8_p \\ \Xi GS;\ \Delta LS \\ \hline pc = t8 \land pc' = t9 \\ \mathit{buffer}'(p) = \mathit{buffer}(p) \frown \langle (W, h) \rangle \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Take}9_p \\ \Xi GS\ \Delta LS \\ \mathit{task}! : \mathit{Task} \\ \hline pc = t9 \land pc' = 1 \\ \mathit{task}! = \mathit{empty} \\ \hline \end{array}$$

The final 4 lines of `take`, corresponding to conflict for the final task in the deque, are specified as follows. Note that the Z schemas corresponding to the CAS, $\mathit{Take}11t_p$ and $\mathit{Take}11f_p$, include a fence (modelled by $\mathit{buffer}(p) = \langle\,\rangle$). Again, 1 is added to the index of the output task in $\mathit{Take}13_p$.

$Take10_p$

$$\begin{array}{l} \Xi GS;\ \Delta LS \\ \hline pc = t10 \wedge pc' = t11 \\ buffer'(p) = buffer(p) \frown \langle (W, h+1) \rangle \end{array}$$

$Take11t_p$

$$\begin{array}{l} \Xi GS;\ \Delta LS \\ \hline buffer(p) = \langle\,\rangle \\ pc = t11 \wedge pc' = t12 \\ H \neq h \end{array}$$

$Take11f_p$

$$\begin{array}{l} \Xi GS;\ \Delta LS \\ \hline buffer(p) = \langle\,\rangle \\ pc = t11 \wedge pc' = t13 \\ H = h \wedge H' = h+1 \end{array}$$

$Take12_p$

$$\begin{array}{l} \Xi GS;\ \Delta LS \\ task! : Task \\ \hline pc = t12 \wedge pc' = 1 \\ task! = empty \end{array}$$

$Take13_p$

$$\begin{array}{l} \Xi GS;\ \Delta LS \\ task! : Task \\ \hline pc = t13 \wedge task! = \overline{tasks(t \bmod W + 1)}(p) \wedge pc' = 1 \end{array}$$

We can model the operation `steal` in a similar fashion. Details are omitted here. Finally, we specify the flush operation. When the identifier of a value to be flushed is in the range $0..W-1$, we add 1 to it to get the corresponding position in the sequence *tasks* which needs to be updated.

$Flush_{cpu}$

$$\begin{array}{l} \Delta GS \\ p! : P \\ \hline buffer(p!) \neq \langle\,\rangle \Rightarrow \\ \quad (\exists\, id : Id;\ val : Task \cup \mathbb{N} \bullet \\ \qquad buffer(p!) = \langle (id, val) \rangle \frown buffer'(p!) \wedge \\ \qquad (id \in 0\,..\,W-1 \Rightarrow tasks' = tasks \oplus \{id+1 \mapsto val\} \wedge T' = T) \wedge \\ \qquad (id = W \Rightarrow T' = val \wedge tasks' = tasks)) \\ buffer(p!) = \langle\,\rangle \Rightarrow tasks' = tasks \wedge T' = T \wedge buffer'(p!) = buffer(p!) \end{array}$$

5.3 Refined Abstract Specification

It is not possible to directly apply the proof method of Sect. 4 to these abstract and concrete specifications of the Chase-Lev deque. The buffers of the coarse-grained abstraction would contain entries for tasks in the sequence *tasks*, but would not contain entries for the concrete variable T which does not appear in the abstract specification. In general, the state representation at the abstract and concrete levels can differ significantly, resulting in a mismatch between buffer values and the number of flushes required for a given operation. To overcome this problem, we introduce a second abstract specification which is a data refinement of the original. This refined abstract specification, like the original, does not have buffers or flushes, but unlike the original has the same state representation as the concrete specification.

The state schema of this refined abstract specification has variables T and H of type $\mathbb{N}$ and a sequence of tasks of length W.

$$\begin{array}{|l}\hline AS1 \\ tasks : \mathrm{seq}\, Task \\ H, T : \mathbb{N} \\ \hline \# tasks = W \\ \hline\end{array} \qquad \begin{array}{|l}\hline AS1Init \\ AS1 \\ \hline H = 0 \land T = 0 \\ \hline\end{array}$$

The Put_p operation adds a task at position ($T \mod W + 1$) of the sequence (($T \mod W$) of the modelled array) and increments T. In the case where there are already W tasks in *tasks*, this will result in the earliest added task to be overwritten (as is done in the implementation). Recall that the behaviour of the original Put_p operation is undefined when Put_p occurs and there are already W tasks.

$$\begin{array}{|l}\hline Put_p \\ \Delta AS1 \\ task? : Task \\ \hline tasks' = tasks \oplus \{(T \bmod W + 1) \mapsto task?\} \\ T' = T + 1 \\ \hline\end{array}$$

The $Take_p$ operation returns *empty* when $H = T$, returns the task at position $(T - 1) \mod W + 1$ of *tasks* and increments T when $H + 1 < T$, and returns the task at position $H \mod W + 1$ of *tasks* and increments H when $H + 1 = T$. There is no conflict with a thief at this level of abstraction as $Take_p$ and $Steal_q$ are atomic.

$$\begin{array}{|l}\hline Take_p \\ \Delta AS1 \\ task! : Task \cup \{empty\} \\ \hline H = T \Rightarrow task! = empty \\ H < T - 1 \Rightarrow task! = tasks((T-1) \bmod W + 1) \land T' = T - 1 \\ H = T - 1 \Rightarrow task! = tasks(H \bmod W + 1) \land H' = H + 1 \\ \hline\end{array}$$

The $Steal_q$ operation is modelled in a similar fashion, details are omitted here. This specification can readily be shown to be a data refinement of the original abstract specification using the simulation rules for Z refinement [7] and the following retrieve relation. This relates the tasks starting from position 1 in *tasks* of AS with those starting from position $H \mod W + 1$ in *tasks* of $AS1$.

$$\begin{array}{|l}
\underline{\;R\;}\\
AS\\
AS1[tasks1/task]\\
\hline
tasks = \{i : 0\,..\,(T - H) - 1 \bullet (i + 1, tasks1((H + i) \bmod W + 1))\}\\
\hline
\end{array}$$

5.4 Coarse-Grained Abstraction

Given the refined abstract specification above, we are now in a position to develop the coarse-grained abstraction. In this case, adding a buffer for each process to $AS1$ results in the schema GS. Hence, we have $BS \mathrel{\widehat=} GS$ and $BSInit \mathrel{\widehat=} GSInit$, and the $Flush_p$ operations is as defined for the concrete specification.

The specification of Put_p needs to take into account that flushes may occur between the two writes. Hence, it is possible that the operation returns with only its last write (to T) in the buffer and the sequence *tasks* already updated.

$$\begin{array}{|l}
\underline{\;Put_p\;}\\
\Delta BS\\
task? : Task\\
\hline
buffer'(p) = buffer(p) \frown \langle (T \bmod W, task?), (W, T + 1)\rangle\\
\lor\\
buffer'(p) = \langle (W, T + 1)\rangle \land tasks' = tasks \oplus \{T \bmod W + 1, task?\}\\
\hline
\end{array}$$

The nondeterminism in the definition of Put_p ensures linearizability can be proved using the proof method presented in Sect. 2. Any flushes to values not written by the current Put_p operation which are flushed during its execution are linearized to occur before the operation. A flush of the write to *tasks* is also linearized to occur before the operation, but in this case the flush of the coarse-grained abstraction is one that occurs when the buffer is empty; the updating of *tasks* is done by the operation.

In general, such nondeterminism will be required in the coarse-grained abstraction whenever an operation can return with more than one of its writes in the buffer. Hence, it was not required for spinlock where there was at most one write in the buffer at each operation return. It is also not required for the remaining operations of the Chase-Lev algorithm which either end with a fence (and hence have no writes in the buffer) or, in the case of a `take` when the buffer is empty, end with one write in the buffer.

To specify $Take_p$, we need to pay careful attention to where the fences occur in the implementation. The fence at line `3` ensures that all values in the buffer at the start of the operation are flushed (specified by the first line of the predicate below). When $H = T$, the value of T is changed twice: the first write (line `t2`) is flushed by the fence at line `t3`, and the second (line `t8`) is not. T is also changed twice when $H = T - 1$. However in this case, the CAS (line `11`) clears the buffer ensuring T is equal to its second value (which happens to be the value of T before the operation).

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Take}_p} \\
\Delta BS \\
\mathit{task}! : \mathit{Task} \\
\hline
(\forall i : 1..W \bullet \mathit{tasks}'(i) = \overline{\mathit{tasks}(i)}(p)) \wedge T' = \overline{T}(p) \\
H = \overline{T}(p) \Rightarrow \mathit{task}! = \mathit{empty} \wedge T' = \overline{T}(p) - 1 \wedge \mathit{buffer}'(p) = \langle H \rangle \\
H < \overline{T}(p) - 1 \Rightarrow \mathit{task}! = \overline{\mathit{tasks}((\overline{T}(p) - 1) \bmod W + 1)}(p) \wedge T' = \overline{T}(p) - 1 \wedge \\
\qquad\qquad\qquad\qquad \mathit{buffer}'(p) = \langle\,\rangle \\
H = \overline{T}(p) - 1 \Rightarrow \mathit{task}! = \overline{\mathit{tasks}(H \bmod W + 1)}(p) \wedge H' = H + 1 \wedge \\
\qquad\qquad\qquad\qquad \mathit{buffer}'(p) = \langle\,\rangle \\
\hline
\end{array}
$$

In the implementation of Steal_q, the CAS at line `s7` causes a fence when $H < T$ (specified by the last line of the predicate below). Since Steal_q and Take_p are still atomic at this level of abstraction, there is no chance of conflict when $H < T$.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Steal}_q} \\
\Delta BS \\
\mathit{task}! : \mathit{Task} \\
\hline
H = \overline{T}(q) \Rightarrow \mathit{task}! = \mathit{empty} \\
H < \overline{T}(q) \Rightarrow \\
\qquad \mathit{task}! = \overline{\mathit{tasks}(H \bmod W + 1)}(q) \wedge H' = H + 1 \\
\qquad (\forall i : 1..W \bullet \mathit{tasks}'(i) = \overline{\mathit{tasks}(i)}(q)) \wedge T' = \overline{T}(q) \wedge \mathit{buffer}(q) = \langle\,\rangle \\
\hline
\end{array}
$$

We are now able to apply our approach to prove that the implementation of Fig. 5 is TSO-linearizable with respect to the refined abstract specification of Sect. 5.3, and hence with respect to the abstract specification of Sect. 5.1.

In other work, Liu et al. [17] suggest a further fence is needed after line `p3` of `put` in order to prove linearizability. This is to prevent a thief process performing a `steal` which returns `empty`, immediately after the worker process has completed a `put` operation on an empty deque, i.e., before the writes of the `put` have been flushed to memory. Since under TSO-linearizability the return of the `put` will be moved to the last flush of the values it writes, the `put` and `steal` operations will overlap in this case and the scenario will linearize to an abstract history where the `steal` which returns `empty` occurs before the `push`.

6 Conclusion

In this chapter we have developed a method by which to simplify proofs of linearizability for algorithms running on the TSO memory model. Instead of having to deal with the effects of both fine-grained atomicity and local buffers in one set of proof obligations, we have used an intermediate specification to partition the proof obligations in two. One set of proof obligations is simply the standard existing notion of linearizability, and any existing proof method could be employed to verify this step. The second set of proof obligations involves verifying that an appropriate transformation (given by *TRANS* defined in Sect. 4) holds.

Although there is existing work on defining linearizability on TSO, to the best of our knowledge this is the first work that provides simplified reasoning for showing how linearizability can be verified for algorithms running on TSO, although mention should be made of the approach in [24] that uses SPIN to check specific runs for TSO-linearizability.

References

1. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and arm multiprocessor machine code. In: Petersen, L., Chakravarty, M.M.T. (eds.) DAMP '09, pp. 13–24. ACM (2008)
2. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007, Volume of 4590 LNCS, pp. 477–490. Springer (2007)
3. Bovet, D., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly, Sebastopol (2005)
4. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (ed.) ESOP 2012, volume 7211 of LNCS, pp. 87–107. Springer (2012)
5. Calcagno, C., Parkinson, M., Vafeiadis,V.: Modular safety checking for fine-grained concurrency. In: Nielson, H.R., Filé, G. (eds.) SAS 2007, volume 4634 of LNCS, pp. 233–238. Springer (2007)
6. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: Gibbons, P.B., Spirakis, P.G. (eds.) SPAA, pp. 21–28. ACM (2005)
7. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications, 2nd edn. Springer, Berlin (2014)
8. Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizability via non-atomic refinement. In: Davies, J., Gibbons, J. (eds.) IFM 2007, volume 4591 of LNCS, pp. 195–214. Springer (2007)
9. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically verified proof obligations for linearizability. ACM Trans. Program. Lang. Syst. **33**(1), 4 (2011)
10. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisabilty with potential linearisation points. In: Butler, M., Schulte, W. (eds.) FM 2011, volume 6664 of LNCS, pp. 323–337. Springer (2011)
11. Derrick, J., Smith, G., Dongol, B.: Verifying linearizability on TSO architectures. In: iFM 2014, volume 8739 of LNCS, pp. 341–356 (2014)
12. Derrick, J., Smith, G., Groves, l., Dongol, B.: Using coarse-grained abstractions to verify linearizability on TSO architectures. In: HVC2014, volume 8855 of LNCS (2014)
13. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Nunez, M. (eds.) FORTE 2004, volume 3235 of LNCS, pp. 97–114. Springer (2004)
14. Dongol, B., Derrick, J.: Verifying linearisability: a comparative survey. ACM Comput. Surv. **48**(2):19:1–19:43 (2015)
15. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: sequentially consistent specifications of TSO libraries. In: Aguilera, M. (ed.) DISC 2012, volume 7611 of LNCS, pp. 31–45. Springer (2012)
16. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)

17. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: Vitek, J., Lin, H., Tip, F. (eds.) PLDI, pp. 429–440. ACM (2012)
18. Morrison, A., Afek, Y.: Fence-free work stealing on bounded TSO processors. In: ASPLOS, pp. 413–426. ACM (2014)
19. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Automated Deduction, pp. 13–39. Kluwer (1998)
20. Schellhorn, G., Wehrheim, H., Derrick, J.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Logic (2014)
21. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)
22. Smith, G., Derrick, J., Dongol, B.: Admit your weakness: Verifying correctness on TSO architectures. In: FACS, volume 8997 of LNCS. Springer (2015)
23. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2011)
24. Travkin, O., Mütze, A., Wehrheim, H.: SPIN as a linearizability checker under weak memory models. In: Bertacco, V., Legay, A. (eds.), HVC2013, volume 8244 of LNCS, pp. 311–326. Springer (2013)
25. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)

Part IV
Interfaces and Linking

Linking Discrete and Continuous Models, Applied to Traffic Manoeuvrers

Ernst-Rüdiger Olderog, Anders P. Ravn and Rafael Wisniewski

Abstract The interplay between discrete and continuous dynamical models is discussed, and a systematic approach to developing and combining these models together is outlined. The combination is done with linking predicates that define refinement relations between the models. As a case study, we build an abstract, discr spatial model and a concrete, continuous dynamic model for traffic manoeuvrers of multiple vehicles on highways. In the discrete model we show the safety (collision freedom) of distance keeping and lane-change manoeuvrers using events and actions to specify state transitions. By linking the discrete and continuous model via suitable predicates that express the discrete events and actions as distances and set-points in the continuous model, the safety carries over to the concrete model.

1 Introduction

Hybrid systems were introduced in order to model dynamical systems with a complex interaction between discrete actions and continuous evolutions in their trajectories [15]. Semantic models in the form of Hybrid Automata with the underlying transition systems [2, 29] were soon developed, and simulation tools like Stateflow [30] and Ptolemy II [24] appeared as well. Due to the success of model checking for timed

This research was partially supported by the German Research Foundation (DFG) in the Trans-regional Collaborative Research Center (SFB/TR 14) AVACS(www.avacs.org). This chapter is a revised and extended version of the conference paper [35].

E.-R. Olderog
Department of Computing Science, University of Oldenburg, Oldenburg, Germany
e-mail: olderog@informatik.uni-oldenburg.de

A.P. Ravn (✉)
Department of Computer Science, Aalborg University, Aalborg, Denmark
e-mail: apr@cs.aau.dk

R. Wisniewski
Department of Automation, Aalborg University, Aalborg, Denmark
e-mail: raf@es.aau.dk

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_5

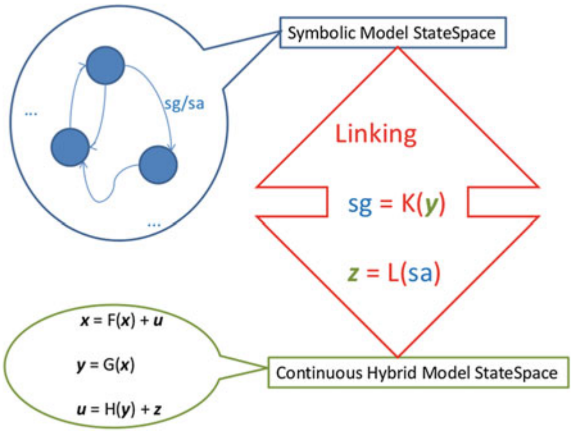


Fig. 1 Modelling approach. The discrete model is a collection of discrete automata with transitions governed by symbolic guards, *sg*, and with symbolic actions, *sa*. The underlying symbolic state space is hybrid with time evolutions. However, it is asserted that time steps do not change the value of guards and actions. The continuous model is a conventional control model which accepts set points, *z*. Linking is given via suitable functions K and L

automata [3], much effort has been directed towards analysis tools which use over- and under-approximations of hybrid automata [12, 14, 51], because it was clear from the outset that decidability was impossible even for very simple models.

There has been much progress both in analysis tools and in the amount of case studies, but it is still hard to find general composition principles. Often a system is decomposed into simpler subsystems that are loosely coupled [4, 20, 42] and thus can be analyzed individually. This loose coupling among concurrently operating subsystems was illustrated in [7], and it was analysed at a semantic level for heterogeneous subsystems in [38]. One observation though is that verification is usually done on subsystem models abstracted from detailed continuous models. It is this decomposition that is in the focus of this work. In a search for a more general and perhaps even teachable ap1proach to performing this abstraction, we have tried to extract the principles from our continued efforts in modelling and verifying vehicle manoeuvrers in traffic, because it is a setting with a complex state space, a demand for decentralized control, and hybrid behaviours.

Here, we have reached the conclusion that a key point in the abstraction is to keep the discrete part, often a supervisory layer, on symbolic and finite level without any direct reference to time, because it allows for exhaustive verification using conventional techniques. However, this will in itself leave the continuous dynamics as an unexplored postulate. Thus, there is a need for linking the symbolic quantities of the discrete model to the concrete continuous model by a proper refinement relation. Via linking, behavioural properties of the abstract model are preserved for the concrete model. An inspiration is here the data refinement relations explored in program verification [8]. However, in the reactive setting, linking predicates as in the approach

of UTP (Unifying Theories of Programming) [23] are more suitable. In summary, the approach presented has the following steps, illustrated in Fig. 1.
For the symbolic, discrete model:

1. build a qualitative model of the context with symbolic representation of states of objects.,
2. formulate rules for interaction as finite state machines operating on the symbolic state (If the state machines use communication protocols, timeout transitions may be used to compensate for lost messages),
3. specify safety and liveness properties of the symbolic state,
4. verify the properties.

These steps are illustrated on the case of vehicle manoeuvrers in Sects. 2 and 3.

When this part has gone through a number of iterations and the result is satisfactory, consider the concrete model:

5. identify a concrete dynamical model for the objects including available or at least plausible sensors and actuators,
6. link the models by relating the symbolic state variables to concrete observables that are computed by a controller for the dynamical system using available sensors and concrete models of the individual objects, also link symbolic actions to set points for the control,
7. design and validate the controllers and observers.

These steps are illustrated on the case in Sects. 4–6.

Note that often the two models may develop concurrently. When this happens, it is important to keep the linkage stable when doing separate iterations.

A pragmatic consideration when designing the linking in the concrete case has been to design a system where a smart car can navigate among ordinary dumb cars. There is no need to require all cars to be smart and able to communicate with other cars. This has implications for the sensors and actuators, see Fig. 3 in Sect. 5, as well as impact on the symbolic guards and actions.

In Sect. 7, we comment on generally related work, while the conclusion in Sect. 8 considers limitations of the approach and potential for tool support.

2 Symbolic Model

In this section, we summarize and adapt the model of [22]. In this model, a multilane highway has an infinite extension with positions represented by real numbers in $\mathbb{R}$ and with lanes represented by a finite set of natural numbers, $\mathbb{L} = \{0, \ldots, N\}$. We assume that all traffic proceeds in one direction, with increasing position values, in pictures shown from left to right. The highway is populated by cars with unique identities denoted by capital letters $\mathbb{I} = \{A, B, \ldots\}$.

At each moment in time, we represent the traffic on the highway by a *traffic snapshot*. It records for each car the current position *pos* (at the rear end of the car) and speed *spd*, and on which lanes the car *reserves* or *claims* space. The idea is that a reserved space is owned by a unique car. Thus for safety, we have to show that reserved spaces of different cars are mutually exclusive. In contrast, a claimed space is used in preparation of a lane change and may still overlap with claimed or reserved spaces of other cars. However, then the lane change must not take place. The length of reserved and claimed spaces is given by the *safety distance*, which is the length of the car plus a safe estimate of the (speed-dependent) braking distance that the car will need to come to a complete standstill.

Definition 1 A *traffic snapshot* $\mathscr{T}$ comprises the functions pos, spd, res, clm

- $pos : \mathbb{I} \to \mathbb{R}$ such that $pos(C)$ is the position of car C along the lanes,
- $spd : \mathbb{I} \to \mathbb{R}$ such that $spd(C)$ is the current speed of the car C,
- $res : \mathbb{I} \to \mathscr{P}(\mathbb{L})$ such that $res(C)$ is the set of lanes C reserves,
- $clm : \mathbb{I} \to \mathscr{P}(\mathbb{L})$ such that $clm(C)$ is the set of lanes C claims.

We denote the set of all traffic snapshots by $\mathbb{T}$.

Note that in $\mathscr{T}$, it is not specified which space is occupied on the reserved and claimed lanes. This information is given by an uninterpreted function se for *safety envelope*. For a given traffic snapshot $\mathscr{T}$, we introduce for each car C its *safety envelope* $se_{\mathscr{T}}(C)$ as the interval $se_{\mathscr{T}}(C) = [pos(C), pos(C) + d(C)]$ starting at the current position $pos(C)$ of the car and of some uninterpreted length $d(C) > 0$, which is intended to be the safety distance of car C dependent on its current speed $spd(C)$. The exact value of $d(C)$ is not known in the symbolic model, but will be determined in the concrete dynamic model.

2.1 View

For our safety proof, we restricst ourselves to finite parts of a traffic snapshot $\mathscr{T}$ called *views*; the intuition being that safety depends on local information only.

Definition 2 A *view* $V = (\mathrm{L}, \mathrm{X}, E)$ consists of an interval of lanes visible in the view, $\mathrm{L} = [l, n] \subseteq \mathbb{L}$, and the extension visible in the view, $\mathrm{X} = [r, t] \subseteq \mathbb{R}$, and $E \in \mathbb{I}$, the identifier of the car under consideration.

A *subview* of V is obtained by restricting the lanes and extension we observe. For this we use sub- and superscript notation: $V^{\mathrm{L}'} = (\mathrm{L}', \mathrm{X}, E)$ and $V_{\mathrm{X}'} = (\mathrm{L}, \mathrm{X}', E)$, where L' and X' are subintervals of L and X, respectively.

For a car E and a traffic snapshot $\mathscr{T} = (pos, spd, res, clm)$ its *standard view* is

$$V_s(E, \mathscr{T}) = (\mathbb{L}, [pos(E) - ho, pos(E) + ho], E) \ ,$$

where the *horizon ho* is chosen such that a car driving at maximum speed can, with lowest deceleration, come to a standstill within the horizon.

2.2 Spatial Logic

To specify properties of traffic snapshots within a given view in an intuitive and yet precise way, we use a two-dimensional spatial interval logic, MLSL (Multi-Lane Spatial Logic) [22]. In this logic, the horizontal dimension is continuous, representing positions on a highway, and the vertical dimension is discrete, representing the number of a lane on a highway. In the syntax, variables ranging over car identifiers are denoted by small letters c, d, u and v. To refer to the car owning the current view, we use a special variable *ego*. By Var we denote the set of all these variables. Additionally, the letter γ ranges over car identifiers or elements in Var.

Definition 3 *(Syntax)* The syntax of the *multi-lane spatial logic MLSL* is given by the following formulae:

$$\phi ::= true \mid u = v \mid free \mid re(\gamma) \mid cl(\gamma) \mid$$

$$\phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \exists v\colon \phi_1 \mid \phi_1 \frown \phi_2 \mid \begin{matrix}\phi_2\\ \phi_1\end{matrix}$$

We denote the set of all MLSL formulae by Φ.

Formulae of MLSL express the spatial status of neighbouring lanes on a multi-lane highway. For a lane, the spatial status describes whether parts of it are reserved or claimed by a car or completely free. To this end, MLSL has atoms $re(\gamma)$, $cl(\gamma)$, and *free*, and two chop operators: the horizontal chop $\phi_1 \frown \phi_2$ expresses that an interval can be divided into two horizontally adjacent parts such that ϕ_1 holds in the left part and ϕ_2 in the right part, and the vertical chop $\begin{matrix}\phi_2\\ \phi_1\end{matrix}$ expresses that an interval can be divided into two vertically adjacent parts where ϕ_1 holds on the lower part and ϕ_2 on the upper part. We use juxtaposition for the vertical chop to have a correspondence to the visual layout in traffic snapshots.

The logic is given a semantics that defines the when traffic snapshots satisfy a given formula.

Definition 4 *(Semantics)* The *satisfaction* $\models$ of formulae is defined inductively with respect to a *model* $\mathcal{M} = (\mathcal{T}, V, \nu)$ comprising a traffic snapshot $\mathcal{T}$, a view $V = (\mathrm{L}, \mathrm{X}, E)$ with $\mathrm{L} = [l, n]$ and $\mathrm{X} = [r, t]$, and a valuation $\nu : \mathbb{I} \cup \mathrm{Var} \to \mathbb{I}$ *consistent* with V, i.e., with $\nu(ego) = E$ and $\nu(C) = C$ for $C \in \mathbb{I}$:

$$
\begin{array}{lll}
\mathcal{M} \models true & & \text{for all } \mathcal{M} \\
\mathcal{M} \models u = v & \Leftrightarrow & \nu(u) = \nu(v) \\
\mathcal{M} \models free & \Leftrightarrow & |\mathrm{L}| = 1 \text{ and } |\mathrm{X}| > 0 \text{ and} \\
 & & \forall C \in \mathbb{I}\colon \mathrm{L} \subseteq res(C) \cup clm(C) \Rightarrow se_{\mathcal{T}}(C) \cap (r, t) = \emptyset \\
\mathcal{M} \models re(\gamma) & \Leftrightarrow & |\mathrm{L}| = 1 \text{ and } |\mathrm{X}| > 0 \text{ and} \\
 & & \mathrm{L} \subseteq res(\nu(\gamma)) \text{ and } \mathrm{X} \subseteq se_{\mathcal{T}}(\nu(\gamma))
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{M} \models cl(\gamma) & \Leftrightarrow & |\mathrm{L}| = 1 \text{ and } |\mathrm{X}| > 0 \text{ and } \mathrm{L} \subseteq clm(\nu(\gamma)) \text{ and } \mathrm{X} \subseteq se_{\mathcal{T}}(\nu(\gamma)) \\
\mathcal{M} \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \mathcal{M} \models \phi_1 \text{ and } \mathcal{M} \models \phi_2 \\
\mathcal{M} \models \neg\phi & \Leftrightarrow & \text{not } \mathcal{M} \models \phi \\
\mathcal{M} \models \exists v \colon \phi & \Leftrightarrow & \exists\, \alpha \in \mathbb{I}\colon (\mathcal{T}, V, \nu \oplus \{v \mapsto \alpha\}) \models \phi \\
\mathcal{M} \models \phi_1 \frown \phi_2 & \Leftrightarrow & \exists\, s : r \leq s \leq t \text{ and} \\
 & & (\mathcal{T}, V_{[r,s]}, \nu) \models \phi_1 \text{ and } (\mathcal{T}, V_{[s,t]}, \nu) \models \phi_2 \\
\mathcal{M} \models \begin{matrix}\phi_2 \\ \phi_1\end{matrix} & \Leftrightarrow & \exists\, m : l - 1 \leq m \leq n + 1 \text{ and} \\
 & & (\mathcal{T}, V^{[l,m]}, \nu) \models \phi_1 \text{ and } (\mathcal{T}, V^{[m+1,n]}, \nu) \models \phi_2
\end{array}
$$

We write $\mathcal{T} \models \phi$ if $(\mathcal{T}, V, \nu) \models \phi$ for all views V and consistent valuations ν.

For the semantics of the vertical chop, we set the interval $[l, m] = \emptyset$ if $l > m$. A view V with an empty set of lanes satisfies only *true* or an equivalent formula. Both chop modalities are associative. Other logical operators like $\vee$, $\rightarrow$, $\leftrightarrow$ and $\forall$ are treated as abbreviations. Also, we use the notation $\langle\phi\rangle$ for the two-dimensional modality *somewhere* ϕ, defined in terms of both chop operators:

$$\langle\phi\rangle \equiv true \frown \begin{pmatrix} true \\ \phi \\ true \end{pmatrix} \frown true.$$

For example, $Safe \equiv \forall c, d : c \neq d \rightarrow \neg \langle re(c) \wedge re(d)\rangle$ expresses the safety property that any two different cars have disjoint reserved spaces.

2.3 Transition System

A traffic snapshot is an instant picture of the highway traffic. The following *transitions* describe how it may change. Time may pass or a car may perform several actions when attempting and performing a lane change. We use the overriding notation $\oplus$ for function updates [46].

$$\mathcal{T} \xrightarrow{t} \mathcal{T}' \quad \Leftrightarrow \quad \begin{array}{l} \mathcal{T}' = (pos', spd', res, clm) \\ \wedge\, \forall C \in \mathbb{I}\colon pos'(C) > pos(C) \end{array} \tag{1}$$

$$\mathcal{T} \xrightarrow{\mathsf{c}(C,n)} \mathcal{T}' \quad \Leftrightarrow \quad \begin{array}{l} \mathcal{T}' = (pos, spd, res, clm') \\ \wedge\, |clm(C)| = 0 \wedge |res(C)| = 1 \\ \wedge\, \{n+1, n-1\} \cap res(C) \neq \emptyset \\ \wedge\, clm' = clm \oplus \{C \mapsto \{n\}\} \end{array} \tag{2}$$

$$\mathcal{T} \xrightarrow{\mathsf{wd\ c}(C)} \mathcal{T}' \quad \Leftrightarrow \quad \begin{aligned} &\mathcal{T}' = (pos, spd, res, clm') \\ &\wedge clm' = clm \oplus \{C \mapsto \emptyset\} \end{aligned} \tag{3}$$

$$\mathcal{T} \xrightarrow{\mathsf{r}(C)} \mathcal{T}' \quad \Leftrightarrow \quad \begin{aligned} &\mathcal{T}' = (, pos, spd, res', clm') \\ &\wedge clm' = clm \oplus \{C \mapsto \emptyset\} \\ &\wedge res' = res \oplus \\ &\quad \{C \mapsto res(C) \cup clm(C)\} \end{aligned} \tag{4}$$

$$\mathcal{T} \xrightarrow{\mathsf{wd\ r}(C,n)} \mathcal{T}' \quad \Leftrightarrow \quad \begin{aligned} &\mathcal{T}' = (pos, spd, res', clm) \\ &\wedge res' = res \oplus \{C \mapsto \{n\}\} \\ &\wedge n \in res(C) \wedge |res(C)| = 2. \end{aligned} \tag{5}$$

In (1), time passes, which results in the cars moving along the highway to the right. However, note that reservations, *res*, and claims, *clm*, cannot change during time passing transitions. The new position and speed of each car is determined by the dynamics of them, which is described at the concrete level. A car may *claim* a neighbouring lane n (2) if and only if it does not already claim a lane or is in the progress of changing the lane and therefore reserves two lanes. Furthermore, a car may *withdraw* a claim (3) or *reserve* a previously claimed lane (4) or withdraw the reservation of all but one of the lanes it is moving on (5).

3 Abstract Controllers

In this section we present abstract car controllers for keeping distance and changing lanes. By abstract, we mean that properties, invariants and guards of transitions are given by MLSL formulas. The controllers should guarantee that at any moment the spaces reserved by different cars are disjoint. This is expressed concisely by

$$Safe \equiv \forall c, d : c \neq d \Rightarrow \neg \langle re(c) \wedge re(d) \rangle ,$$

stating that in any lane any two different cars have disjoint reserved spaces. The quantification over lanes arises implicitly by the negation of the somewhere modality in *Safe*. A traffic snapshot $\mathcal{T}$ is *safe* if $\mathcal{T} \models Safe$ holds.

3.1 Keeping Distance

A distance controller DC of a car E should guarantee the safety as long as E is driving along the highway without making any new claim or reservation. This is expressed by time transitions among traffic snapshots: $\mathcal{T} \xrightarrow{t} \mathcal{T}'$. From the perspective of the car E, safety means that the following *collision check* remains false:

$$cc \equiv \exists c\colon c \neq ego \wedge \langle re(ego) \wedge re(c)\rangle\,.$$

Thus we require:
(DC) The distance controller DC of a car E keeps the property $\neg cc$ invariant under time transitions, i.e., for every transition $\mathcal{T} \xrightarrow{t} \mathcal{T}'$ whenever $\mathcal{T} \models \neg cc$, also $\mathcal{T}' \models \neg cc$.

3.2 Changing Lanes

We specify an abstract controller by a *timed automaton* [3] with clocks ranging over $\mathbb{R}_{\geq 0}$ and data variables ranging over $\mathbb{L}$ and $\mathbb{I}$. Strictly speaking, the single clock x, which is used in the automaton, is unnecessary for proving safety; it is added to ensure liveness. MLSL formulae appear in transition guards and state invariants. This can be seen in the lane-change controller in Fig. 2, where the MLSL formulae ϕ_1 and ϕ_2 are kept symbolic. The abstract lane-change controller LCP of [22] is an instantiation of this controller, except that it has the invariant $\neg cc$ in the initial state q_0. Here this property is ensured invariantly by the distance controller DC.

LCP assumes that every car, E, knows the full extension of claims and reservations of all cars within its view. It thus has *perfect knowledge* of its neighbouring cars (hence the letter P in the name of the controller); E perceives another car C as soon as C's safety envelope enters the view of E. In the following and in Sect. 5, we identify the car variables *ego* and c with their values, the cars E and C, respectively.

At the initial state q_0 of LCP, the car has reserved exactly one lane, which is saved in the variable n. An auxiliary variable l stores the lane the *ego* car wants to move to. Suppose *ego* intends to change to a neighboring lane, then it adheres to the following protocol. First, it claims a space on the target lane adjacent to and of the same extension as the reservation on its current lane, moving to state q_1. Subsequently, it checks for a *potential collision* (pc), i.e., whether its claim intersects with the reservation or claim of any other car. This is expressed by the MLSL formula

$$pc \equiv \exists c : c \neq ego \wedge \langle cl(ego) \wedge (re(c) \vee cl(c))\rangle\,.$$

If pc occurs, *ego* withdraws its claim and returns to state q_0, giving up the wish to change lanes for the moment. Otherwise, *ego* turns its claim into reservations and thus reserves two lanes. This is in state q_3, During this double reservation *ego*

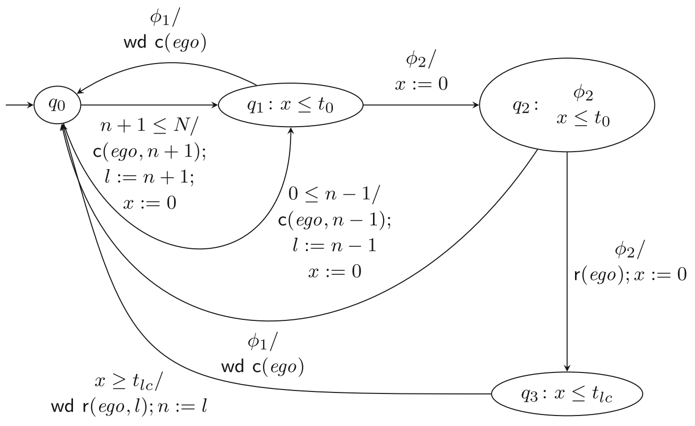


Fig. 2 The lane-change controller LCP with $\phi_1 \equiv pc$ and $\phi_2 \equiv \neg pc$

changes lane within t_{lc} time units, an upper time bound for the lane change. Then *ego* withdraws its reservation on the original lane and continues to drive on the target lane, being again in state q_0. In this protocol, only turning the claim into a reservation (in the transition from state q_2 to state q_3) may violate the safety property. Thus in LCP of Fig. 2, we instantiate $\phi_1 \equiv pc$ and $\phi_2 \equiv \neg pc$.

In order to ensure liveness in the states q_0 and q_1, they are to be left within t_0 time units. Liveness in state q_0 could be ensured by adding an invariant asserting that the state should be left when a claim is made. The lane change timeout t_{lc} should strictly speaking be replaced by a symbolic guard that would be asserted by the concrete model when a lane change was completed. This symbolic guard would then be linked to either a sensor value or most likely to a timer in the concrete model.

3.3 Safety

We stipulate now that every car is equipped with the controllers DC and LCP (or that its driver manually follows its protocol). Under these assumptions, we can show:

Theorem 1 (Safety of DC and LCP) *Let $\mathcal{T}_0$ be an initial safe traffic snapshot. Then every traffic snapshot $\mathcal{T}$ that is reachable from $\mathcal{T}_0$ by transitions allowed by the controllers DC and LCP is safe.*

Proof As in [22], we fix an arbitrary car E and shows that $\neg cc$ holds for every traffic snapshot $\mathcal{T}$ reachable from $\mathcal{T}_0$. The argument is by induction on the number of transitions needed to reach $\mathcal{T}$ from $\mathcal{T}_0$, and the crucial case in the induction step is that of the reservation transition. In contrast to [22], the initial state q_0 of LCP in Fig. 2 does not have $\neg cc$ as a built-in invariant. However, since the distance controller DC is running in parallel to LCP, the safety property $\neg cc$ is an invariant for this state. Moreover, it is also invariant under any transition that is not creating any new reservation. Regarding LCP, we thus have that $\neg cc$ holds in the start state q_2 of the reservation transition from state q_2 to state q_3 in LCP. As in [22], it can be shown that performing the reservation transition in state q_2 satisfying both $\neg cc$ and $\neg pc$ leads to q_3 satisfying $\neg cc$. $\square$

4 Concrete Model

The aim of this section is to present a physical model of a vehicle, which describes the position $pos(C)$ and the speed $spd(C)$ of a vehicle C. It will lay the basis for the controller design in Sect. 6.

4.1 Longitudinal Motion

A vehicle C is characterised by its velocity given in $[m/s]$ at the current time t given in $[s]$, $v_C : \mathbb{R}_+ \to \mathbb{R}_+$. Both the time and the velocity are considered non-negative reals. The acceleration and braking of the vehicle C is realised by a torque $T \equiv T_C : \mathbb{R}_+ \to \mathbb{R}$ given in $[Nm]$. The torque is applied to the wheels from the transmission and braking system, and it belongs at any given time to an interval $[\underline{T}, \overline{T}] \equiv [\underline{T_C}, \overline{T_C}]$, where $\underline{T_C} < 0$ is the maximal torque of the brakes, and $\overline{T_C} > 0$ is the torque at full throttle.

To model aerodynamic drag force, we introduce a drag coefficient C_W. The drag force is proportional to the square of the velocity

$$C_\mathrm{W}(t) v_C^2(t).$$

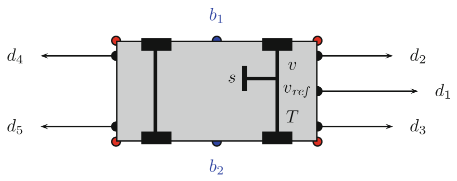
As indicated in the above equation, C_W varies in time. Specifically, C_W is characterised as follows. Suppose a vehicle D drives in front of the vehicle under consideration C. The drag coefficient is an empirical quantity approximated by

$$C_\mathrm{W}(\delta, v_D) = C_C \left(1 - \exp\left(-\frac{a\delta}{C_D v_D}\right)\right)^2,$$

where C_C, C_D are the aerodynamic coefficients of the vehicles C and D, and a is a constant [47]. In short, the aerodynamic coefficient of a vehicle is determined by its geometry: shape and size. The drag coefficient is positive, $\mathrm{Image}(C_\mathrm{W}) \subseteq [0, C_C]$. It converges to C_C for small distances δ and large velocities v_D.

As a consequence, the dynamics of the vehicle C is given by

$$(Mr^2 + J)\dot{v}_C(t) = -C_\mathrm{W}(\delta(t), v_D(t)) r^2 v_C(t)^2 + rT(t),$$



We assume that each car is equipped with the following observers:

- $\hat{v}$ gives its own velocity,
- $\hat{d}_1$ gives the distance to the car ahead in the same lane,
- $\hat{d}_2$ ($\hat{d}_3$) give the distance to the car ahead in the left (right) neighboring lane,
- $\hat{d}_4$ ($\hat{d}_5$) give the distance to the car behind in the left (right) neighboring lane, and
- $\hat{b}_1$ ($\hat{b}_2$) tell whether a car on the lane next to the left (right) one is "blinking", indicating a desired lane change to the left (right) neighboring lane.

Also, each car has its blinkers (here shown as small circles at the four corners of the car) and a torque T as actuators. Steering s and desired reference velocity v_{ref} are inputs from the driver.

Fig. 3 Car with observers and actuators

where M is the mass of the vehicle C $[kg]$, J is the combined moments of inertia of the wheels $[kgm^2]$, and r is the radius of the wheels $[m]$.

Let X be the state space of the vehicle C (with the vehicle D driving in in front). It is the linear space of vectors comprising of the velocity v_C of the vehicle C, and the distance δ from C to D, i.e., $X = \mathbb{R}^2$. We assume that both the velocity and the distance are available as indicated in Fig. 3, where sensor $\hat{v}$ measures v_C and $\hat{d}_1$ measures δ. If the vehicle D is out of range the distance sensor delivers the value ∞.

A feedback controller is a function $T : X \to [\underline{T}, \overline{T}]$ that takes the current state to the torque. Negative values are realised by the braking system; whereas, the positive values are realised by the transmission (the throttle). As a consequence, $T(t) = T(v_C(t), \delta(t))$.

To simplify the notation, we introduce

$$\begin{aligned} x(t) = (x_1(t), x_2(t)) &\equiv (\delta(t), v_C(t)) \in \mathbb{R}^2 \\ z(t) &\equiv v_D(t) \in \mathbb{R} \\ b &\equiv \frac{r}{Mr^2 + J} \in \mathbb{R} \\ a(x_1, z) &\equiv rbC_{\mathrm{W}}(x_1, z) \in C^\infty(\mathbb{R}^2, \mathbb{R}_+) \\ u(t) &\equiv bT \in \mathbb{R} \\ (-\underline{u}, \overline{u}) &\equiv (-b\underline{T}, b\overline{T}) \in \mathbb{R}^2_+ \\ x_0 &\equiv (d^0, v_C^0) \in \mathbb{R}^2. \end{aligned} \tag{6}$$

As a result, the equations of motion are given by the following Cauchy problem with $x(0) = x_0$:

$$\begin{aligned}\dot{x}_1(t) &= z(t) - x_2(t)\\ \dot{x}_2(t) &= -a(x_1(t), z(t))\, x_2(t)^2 + u(t),\end{aligned} \tag{7}$$

where $u(t) \in [\underline{u}, \overline{u}]$. The subscripts of x refer to the components of the vector x.

Remark 1 The Eq. (7) can be used to compute the safety or braking distance $d_s(v_c^0)$ as a function of the initial velocity v_c^0 of the vehicle C. To this end, let $z(t) = 0$, i.e., the vehicle in front instantaneously stops

$$\dot{x}_1(t) = -x_2(t) \text{ and } \dot{x}_2(t) \le \underline{u}$$

for $x_0 = (0, v_C^0)$. To compute the braking distance, we apply the Gronwall lemma [48], which we state now for completeness. Suppose that k is a non-negative and bounded function on an interval $[t_0, t_1]$ and l a non-decreasing function on the same interval.

If

$$v(t) \le l(t) + \int_{t_0}^{t} k(s)v(s)ds$$

for $t \in [t_0, t_1]$, then

$$v(t) \le \exp\left(\int_{t_0}^{t} k(s)ds\right) l(t).$$

Consequently, by the Gronwall lemma, the time to stop is $t \le \hat{t} \equiv -\frac{v_C^0}{\underline{u}}$ (notice that $\underline{u} < 0$). Hence, the braking distance is at most $d_s(v_C^0) = -\frac{(v_C^0)^2}{2\underline{u}}$.

4.2 Lateral Motion

So far, we have not discussed lateral motion. For the details of modelling, we refer to [37]. In short, the kinematic model of the vehicle C is given by the global position

$$\dot{X} = v_C \cos(\psi + \beta) \tag{8a}$$

$$\dot{Y} = v_C \sin(\psi + \beta), \tag{8b}$$

where v_C is the velocity of the vehicle C, β is the slip angle of the vehicle defined below, and ψ is the yaw angle, that defines the orientation angle of the vehicle w.r.t. the x-axis

$$\dot{\psi} = \frac{v_C}{l} \cos(\beta) \tan(\theta). \tag{9}$$

In (9), l is the vehicle base, the distance between the rear and the front wheels, and θ is the angle between the front wheel and the longitudinal axis of the vehicle, with $\theta \in [\underline{\theta}, \overline{\theta}]$ for $\underline{\theta} < 0$ and $\overline{\theta} > 0$; θ as the control input.

The slip angle of the vehicle is given by the relation

$$\beta \equiv \beta(\theta) = \tan^{-1}\left(\frac{l_{\mathrm{r}}\tan(\theta)}{l}\right),$$

where l_{r} is the distance between the centre of gravity and the rear wheel.

5 Linking

To link the abstract and the concrete model, we must map the symbolic observables and events to observer functions in the controllers. In this work, we assume that each car is equipped with the observers, realised by suitable sensors, and actuators listed in Fig. 3.

The abstract controller LPC takes a view of the traffic snapshot, represented by MLSL formulae built with the atoms *free, re(c),cl(c)*. By Theorem 1, this suffices for the safety check at the abstract level. However, the check *assumes* that the reserved or claimed spaces are large enough. Whether this assumption is true, depends on the concrete controller based on the car dynamics.

5.1 Distance Controller

We first turn to the distance controller DC in each car as formalized by assumption **DC**. Every car E keeps the property $\neg cc$ invariant under time transitions, expressing that "no collision" occurs:

$$\neg cc \equiv \neg\exists c\colon c \neq ego \wedge \langle re(ego) \wedge re(c)\rangle\,.$$

Since the overlap $re(ego) \wedge re(c)$ is symmetric, the distance controller in *ego* must check forward or backward for any other car c. However, considering all cars together, it suffices that each car *ego* checks only that there is "no collision *forward*". Let *c ahead ego* abbreviate the following MLSL formula expressing that car c is immediately ahead of *ego*:

$$c\ ahead\ ego \equiv \begin{pmatrix}\neg re(ego)\\ \wedge\\ \neg re(c)\end{pmatrix} \frown \begin{pmatrix} re(ego)\ \frown\ \neg re(ego)\\ \wedge\\ \neg re(c)\ \frown\ re(c)\ \frown\ \neg re(c)\end{pmatrix}.$$

Then we replace the invariant $\neg cc$ by the following formula:

$$\neg ccf \equiv \neg\exists c\colon c \neq ego \wedge \langle re(ego) \wedge re(c)\rangle \wedge \langle c\ ahead\ ego\rangle\ .$$

We recall the resulting "forward looking" distance controller DC_f. Note that logically $\neg ccf$ in DC_f is *weaker* than $\neg cc$ in DC, admitting more traffic snapshots. However, when all cars check $\neg ccf$ instead of $\neg cc$, safety remains guaranteed. This is formalized as follows. Consider the abstract setting A, where *all* cars are equipped with DC, and the abstract forward setting A_f, where *all* cars are equipped with DC_f.

Proposition 1 (Safety of DC_f) *Every time transition among traffic snapshots permitted in A_f is also permitted in A.*

In the concrete controller, we have the observable d that implements the abstract safety distance function $d(ego)$ for car ego at its current speed. Also, there is the concrete observable $\hat{d}_1$ measuring the distance to the next car c ahead. The formula $\neg ccf$ is satisfied if the inequality $d < \hat{d}_1$ holds. Thus the linking predicate relating the abstract and concrete levels is here

$$\neg ccf \Leftarrow d < \hat{d}_1.$$

Note that the implication indicates that $d < \hat{d}_1$ admits no more traffic snapshots than $\neg ccf$ does.

5.2 *Lane-Change Controller*

To link the abstract lane change controller LCP to the observers at the concrete level, the MLSL formulae appearing as guards in LCP are replaced by suitable comparisons of observer values read at the concrete level.

Since the distance controller DC is running in parallel to LCP, the safety property $\neg cc$ holds as long as the reservation transition from state q_2 to state q_3 in LCP is *not* performed (cf. Fig. 2 and the proof of Theorem 1). Note that we can *weaken* the guard of any transition in LCP, except for this reservation transition, and the altered lane change controller will stay safe. For example, we may even weaken the guard ϕ_1 to $true$. Then a claim can always be withdrawn, but this does not violate safety.

Regarding the reservation transition from state q_2 to state q_3, the controller will stay safe as long as we *strenghten* its guard ϕ_2, which in LCP is given by the formula

$$\neg pc \equiv \neg\exists c : c \neq ego \wedge \langle cl(ego) \wedge (re(c) \vee cl(c))\rangle$$

expressing that no potential collison occurs. To link $\neg pc$ with the concrete controller, we distinguish the cases of reservation and claim of c.

Case 1: $\phi_{re} \equiv \neg\exists c : c \neq ego \wedge \langle cl(ego) \wedge re(c)\rangle$. This formula states that no (other) car c on ego's target lane has a reservation that overlaps with ego's claim.

The car c may be (i) ahead of *ego* (or aligned with *ego*) or (ii) behind *ego*. In subcase (i), the concrete controller looks forward using the observables d giving the safety distance needed for car *ego* at its current speed and $\hat{d}_t$ (with t either 2 or 3) measuring the distance to the next car c in front of *ego* on the *target* lane of its lane change maneuver. The concrete controller checks the inequality $d_s < \hat{d}_t$. In subcase (ii), the concrete controller looks *backward* using the observables $\hat{d}_b$ (with b either 4 or 5) measuring the distance to the next car behind *ego* on the target lane and $d_{s,max}$, the maximal braking distance of any car, i.e., an *overapproximation* of the actual braking distance of that car. The concrete controller checks the inequality $d_{s,max} < \hat{d}_b$. Thus, the linking predicate relating the abstract and concrete levels is in this case

$$\phi_{re} \Leftarrow d < \hat{d}_t \wedge d_{s,max} < \hat{d}_b.$$

Due to the over-approximation in $d_{s,max}$ the check at the concrete level may be *stronger* than necessary, permitting fewer lane changes than $\neg pc$, but it preserves safety.

Case 2 : $\phi_{cl} \equiv \neg\exists c : c \neq ego \wedge \langle cl(ego) \wedge cl(c)\rangle$.

The formula states that no other car c on *ego*'s target lane has a claim that overlaps with *ego*'s claim. Such a car c may only be in a lane *next* to *ego*'s target lane. In this case, the concrete controller checks with its sensor b_t (with t either 1 or 2) on the side of the target lane for a turn signal of some car c on the lane next to the target lane. The formula ϕ_{cl} is satisfied if $\neg b_t$ holds. Thus, the linking predicate relating the abstract and concrete levels is in this case

$$\phi_{cl} \Leftarrow \neg b_t.$$

Summarising, at the concrete level, we instantiate

$$\phi_2 \equiv (d < \hat{d}_t \wedge d_{s,max} < \hat{d}_b) \wedge \neg b_t,$$

which by the linking predicates for ϕ_{re} and ϕ_{cl} implies $\neg pc$ at the abstract level.

For the guards of the two withdrawal transitions from state q_1 to state q_3 and from state q_2 to state q_0 in Fig. 2, we put $\phi_1 \equiv \neg\phi_2$ for the above instantiation of ϕ_2. Thus compared with the abstract controller LCP, the guard ϕ_1 is weakened, permitting more withdrawals, but as argued before, this preserves safety.

Altogether, instantiating in the controller in Fig. 2 the formula ϕ_2 by the distance inequalities and blinker sensor values as stated above and ϕ_1 by its negation, we obtain a concrete lane-change controller that we call LCP$_c$. Consider the abstract setting ALC, where all cars are equipped with LCP, and the concrete setting CLC, where all cars are equipped with LCP$_c$.

Proposition 2 (Safety of LCP$_c$) *Every reservation transition among traffic snapshots permitted in CLC is also permitted in ALC.*

Combining Propositions 1 and 2, we obtain:

Theorem 2 (Safety of DC$_c$ and LCP$_c$) *Let $\mathscr{T}_0$ be an initial safe traffic snapshot. Then every traffic snapshot $\mathscr{T}$ that is reachable from $\mathscr{T}_0$ by transitions allowed by the controllers DC$_c$ and LCP$_c$ is safe.*

6 Concrete Controllers

The main focus in this section will be on the longitudinal motion control. Nonetheless, for completeness we will provide a control for changing the lane.

6.1 Longitudinal Control

We will address the assumptions for the distance controller used in Sect. 5.1 linking the safety to the safety envelope through the variable d. To this end, we propose a sliding mode controller for a vehicle C that maintains the velocity of the vehicle at the reference v_{ref} until the distance d between C and the vehicle D in front is reached. Subsequently, the distance d is kept. If D is out of range of the distance sensor, the controller keeps the velocity at v_{ref}. In the following, we assume that at full throttle, the control $\overline{u}$ is strong enough to overcome the drag. To this end, we notice that $a(x_1, z) \in [0, rbC_C]$ for any $(x_1, z) \in R^2_+$, where the constant b is defined in (6). Let the speed limit be denoted by $\bar{v}$. Consequently, we assume that the maximal control $\overline{u} > rbC_C\bar{v}^2$. By a safe control, we understand a control that keeps the motion of a vehicle safe.

Definition 5 *(Safe Control)* A *safe controller* for the control system (7) and a function $z : \mathbb{R}_+ \to [0, \bar{v}]$ is a function $u : \mathbb{R}^3 \mapsto \mathbb{R}$ such that the solutions of the dynamical system (7) with $u(t) = u(x(t), z(t))$ satisfy the following condition: If $x_1(0) \geq d$, then $x_1(t) > 0$ for all $t \in \mathbb{R}_+$.

In plain words, Definition 5 says that an on-board controller is safe if: whenever the distance from the controlled vehicle to a vehicle in front is initially greater than d then a collision between these two vehicles will never happen.

Proposition 3 (Existence of a safe controller) *Consider the control system* (7) *and a function $z : \mathbb{R}_+ \to [0, \bar{v}]$. Let $0 \leq v_{\text{ref}} < \bar{v}$, $d \equiv d(\bar{v})$, and $\alpha \equiv rbC_C\bar{v}^2$. Suppose that $\underline{u} < 0$. Let $k > 0$, and define two affine maps*

$$L_1(x) \equiv x_2 - v_{\text{ref}},\ L_2(x, z) \equiv z - x_2 + k(x_1 - d), \tag{10}$$

and a polyhedral set

$$P(z) \equiv \{x \in \mathbb{R}^2 |\ L_1(x) \leq 0 \textit{ and} - L_2(x, z) \leq 0\}. \tag{11}$$

Then the control

$$u(x,z) = \begin{cases} \underline{u} \ for \ x \in \mathbb{R}^2 \backslash P(z) \\ \overline{u} \ for \quad x \in P(z) \end{cases} \tag{12}$$

is safe. Furthermore, the following two properties for the vehicle controlled by the u *in* (12) *hold:*

1. *If* $x_2(0) > v_{\text{ref}}$ *then* $x_2(t) < x_2(0)$ *for all* $t \in \mathbb{R}_+$ *and there is* $\tau \in \mathbb{R}^+$ *such that* $x_2(t) \leq v_{\text{ref}}$ *for* $t > \tau$.
2. *Let* $\beta \equiv \inf\{\dot{z}(t) | \ t \in \mathbb{R}_+\}$ *and* $\gamma \equiv \sup\{\dot{z}(t) | \ t \in \mathbb{R}_+\}$. *Suppose that* $\underline{u} < \beta$ *and* $\overline{u} > \alpha + \gamma$, *and assume* $0 < k < \min\{\beta - \underline{u}, \overline{u} - \alpha - \gamma\}/\bar{v}$. *Then*
   a. *Let* $0 \leq x_1(0) < d$, *and suppose that the controller* (12) *is such that* $u(t) = \overline{u}$ *holds on an interval* $[0, \tau]$. *Then* $x_1(t) > x_1(0)$ *for all* $t \in [0, \tau]$.
   b. $\lim_{t\to\infty} x_1(t) = d$.

Proof If $x_1(0) \in \mathbb{R}^2 \backslash P(z)$, then the following holds. There is a family of open intervals $\{(\underline{\tau}_\alpha, \overline{\tau}_\alpha) | \ \alpha \in \Lambda\}$ such that $x(\underline{\tau}_\alpha) \in P(z)$ and if $t \in (\underline{\tau}_\alpha, \overline{\tau}_\alpha)$ then $x(t) \in \mathbb{R}^2 \backslash P(z)$, hence $u(t) = \underline{u}$, and from (7), $x_1(t) > 0$. If $t \in \mathbb{R} \backslash \bigcup_{\alpha\in\Lambda} (\underline{\tau}_\alpha, \overline{\tau}_\alpha)$ then $x(t) \in P(z(t))$, and thus $x_1(t) \geq d$. The last statement follows from the following. If $x(t) \in P(z(t))$, then

$$k(x_1(t) - d) \geq x_2(t) - z(t). \tag{13}$$

And, we consider two cases: $x_2(t) \geq z(t)$ and $z_2(t) < z(t)$. If $x_2(2) \geq z(t)$, then from (13), $x_1(t) \geq d$. If $z_2(t) < z(t)$, then from (7), $x_1(t) \geq x_1(0) \geq d$. Hence, the control (12) is safe.

We prove Property 1 and Property 2 of the proposition. To this end, we observe that for $x \in \mathbb{R}^2 \backslash P(z)$,

$$\dot{L}_1(x,z) = -a(x_1,z)x_2^2 + \underline{u} \leq \underline{u} < 0 \tag{14}$$

$$\begin{aligned} \dot{L}_2(x,z,\dot{z}) &= \dot{z} + a(x_1,z)x_2^2 + k(z - x_2) - \underline{u} \\ &\geq \beta - k\bar{v} - \underline{u} > 0. \end{aligned} \tag{15}$$

whereas, for $x \in P(z)$,

$$\dot{L}_1(x,z) = -a(x_1,z)x_2^2 + \overline{u} \geq -\alpha + \overline{u} > 0 \tag{16}$$

$$\begin{aligned} \dot{L}_2(x,z,\dot{z}) &= \dot{z} + a(x_1,z)x_2^2 + k(z - x_2) - \overline{u} \\ &\leq \gamma + \alpha + k\bar{v} - \overline{u} < 0. \end{aligned} \tag{17}$$

By (14), Property 1 holds.

We will show Property 2.a. To this end, we notice that $u(t) = \overline{u}$ whenever $x(t) \in P_{z(t)}$. We consider two cases $z(t) > x_2(t)$ and $z(t) \leq x_2(t)$. If $z(t) > x_2(t)$ then $\dot{x}_1(t) = z(t) - x_2(t) > 0$ and Property 2.a follows. Suppose that $z(t) \leq x_2(t)$.

Then $0 < k(x_1(t) - d) \geq z(t) - x_2 + k(x_1(t) - d) = L_2(x(t), z(t)) \geq 0$, which is a contradiction.

To show Property 2.b, we observe that by Inequalities (14)–(17), any flow line of (7) intersects the boundary of P at a point say $\tilde{x}$ (transversally), i.e., there is $t_1 \geq 0$ such that $x(t_1) = \tilde{x}$. If $L_1(\tilde{x}) = 0$, then the solution (in a Filippov sense) $x(\cdot)$ is such that $L_1(x(t)) = 0$ for all $t \in [t_1, t_2]$, where t_2 is the time at which $L_2(x(t_2), z(t_2)) = 0$. Subsequently, the Fillipov solution $x(\cdot)$ is such that $L_2(x(t), z(t)) = 0$ for all $t \geq t_2$. As a consequence, $z(t) - x_2(t) + k(x_1(t) - d) = 0$, which is equivalent to

$$\frac{\mathrm{d}}{\mathrm{d}t}(x_1(t) - d) = -k(x_1(t) - d).$$

Hence, $\lim_{t\to\infty} x_1(t) = d$. $\square$

The above proposition shows that there is a control that keeps the distance from the vehicle C to the vehicle in front safe while the velocity of C does not exceed the reference. Also whenever the vehicle C accelerates, $u(t) = \overline{u}$, and initially the distance $x_1(0)$ is less than d then the distance increases, i.e., the traffic situation is no less safe than it was at the beginning. If the distance between C and D was greater than d then there is no future time that they will hit each other.

To avoid discontinuous control and hence abrupt switches between acceleration $\overline{u}$ and deceleration $\underline{u}$, the control (12) can be replaced by a continuous approximation. To this end, we will need an ε-neighbourhood $\partial P^\varepsilon(z)$ of the boundary $\partial P(z)$ of the polyhedral set $P(z)$. Subsequently, in $P\backslash\partial P^\varepsilon(z)$, we will use u equal to $\overline{u}$, in $\mathbb{R}^2\backslash(P(z) \cup \partial P^\varepsilon(z))$, we will use u equal to $\underline{u}$ and in $\partial P^\varepsilon(z))$, we will use the control that is a linear combination of $\overline{u}$ and $\underline{u}$ weighted by the distance to $\partial P(z)$. These constructions will be detailed below. For this purpose, recall the definitions of L_1, L_2 in (10), and P in (11), and consider

$$\begin{aligned}
&\mathbb{L}_1 \equiv L_1^{-1}(0) = \{x \in \mathbb{R}^2 |\ L_1(x) = 0\} \text{ and } \mathbb{L}_{2,z} \equiv \{x \in \mathbb{R}^2 |\ L_2(x, z) = 0\},\\
&\mathbb{H}_1 \equiv \{x \in \mathbb{R}^2 |\ L_1(x) \leq 0\} \text{ and } \mathbb{H}_{2,z} \equiv \{x \in \mathbb{R}^2 |\ -L_2(x, z) \leq 0\}.
\end{aligned}$$

For an $\varepsilon > 0$, we define a map $h : [-\varepsilon, \varepsilon] \to [0, 1]$ by $y \mapsto \frac{1}{2}\left(\frac{1}{\varepsilon}y + 1\right)$. Let $\mathbb{L}_1^\varepsilon$ be the (closed) ε-neighborhood of $\mathbb{L}_1$ (with respect to the Hausdorff metric), $\mathbb{L}_{2,z}^\varepsilon$ be the ε-neighborhood of $\mathbb{L}_{2,z}$, $\mathbb{H}_1^\varepsilon$ be the ε-neighborhood of $\mathbb{H}_1$, and $\mathbb{H}_{2,z}^\varepsilon$ be the ε-neighborhood of $\mathbb{H}_{2,z}$. Furthermore, we define the ε-neighbourhood $P^\varepsilon(z)$ of P by

$$P^\varepsilon(z) \equiv \mathbb{H}_1^\varepsilon \cap \mathbb{H}_{2,z}^\varepsilon.$$

Let $x^i(x) \equiv x - \pi_{\mathbb{L}_i}(x)$ for $i \in \{1, 2\}$, where $\pi_{\mathbb{L}_1}$ and $\pi_{\mathbb{L}_2}$ are the projections on $\mathbb{L}_1$ and $\mathbb{L}_{2,z}$, respectively. For $l \equiv l(x) = \operatorname{argmax}\{|x^i(x)||\ i \in \{1, 2\}\}$ let

$$y(x) = |x^l| \operatorname{sign}(\langle n^l, x^l\rangle),$$

where $\langle\cdot,\cdot\rangle$ is the scalar product on $\mathbb{R}^2$, n^1 and n^2 are the normal vectors to $\mathbb{L}_1(\cdot)$ and $\mathbb{L}_{2,z}(\cdot)$ pointing into P,

$$n^1 = (0, -1), n^2 = (k, -1).$$

Finally, we are able to define the ε-neighbourhood $\partial P^\varepsilon(z)$ of the boundary of $P(z)$

$$\partial P^\varepsilon(z) \equiv P^\varepsilon(z)\backslash(\mathbb{R}^2\backslash(\mathbb{H}_1^\varepsilon \cup \mathbb{H}_{2,z}^\varepsilon)).$$

We define $\bar{h} : P^{-\varepsilon}(z) \to [0, 1]$ by

$$\bar{h}(x) = h(y(x)).$$

The function $\bar{h}$ takes a point x in the ε-neighbourhood of $\partial P(z)$ and delivers a number between 0 and 1 dependent on the distance to $\partial P(z)$: 0 when the distance is ε and x is outside P and 1 when the distance is ε and x is inside P. The control is then

$$u(x, z) = \begin{cases} \underline{u} & \text{for} \quad x \in \mathbb{R}^2\backslash P^\varepsilon(z) \\ (1 - \bar{h}(x))\underline{u} + \bar{h}(x)\bar{u} & \text{for} \quad x \in P^{-\varepsilon}(z) \\ \bar{u} & \text{for} \quad x \in P(z)\backslash P^{-\varepsilon}(z). \end{cases}$$

The parameter ε is to be chosen as a tradeoff between the accuracy of tracking the distance d and "evenness" of the control. The bigger ε is, the more even and less accurate is the control.

6.2 *Lane Change*

The control for lateral motion is discussed in [37]. For completeness of our study, we propose a facile feedforward control for changing the lane. To avoid a collision during the maneuver of changing the lanes, it is assumed that the minimum distance d to the front vehicles in the current lane and the neighboring target lane is big enough, i.e., greater than the sum of the maximal braking distance of the vehicle C and the distance $\int_0^{t_{lc}} v_C(t)dt$ traveled by C during the lane change.

Recall the lateral motion given by the lateral position Y in (8b) and the yaw angle ψ in (9). We will use the notation

$$b(\theta) \equiv b(\theta, v_C) \equiv \frac{v_C}{l}\cos(\beta(\theta))\tan(\theta).$$

The next proposition characterises the the lateral motion

Proposition 4 *Suppose* $b(\theta) \neq 0$. *Then the solution of* (8b) *and* (9) *belongs to the graph* $\Gamma \equiv \{(\psi, Y) \in]-\pi, \pi[\times\mathbb{R}|\ Y = F(\psi)\}$ *of the function*

$$F \equiv F_{\theta,y_0,\psi_0,v_C} : \psi \mapsto \tilde{y}_0(y_0,\psi_0) - \frac{v_C}{b(\theta)}\cos(\psi + \beta(\theta)),$$

where $\tilde{y}_0(y_0,\psi_0) = y_0 + \frac{v_C}{b(\theta)}\cos(\psi_0 + \beta(\theta))$*, and* y_0 *is the initial lateral position, and hence* ψ_0 *is the initial yaw angle.*

Proof The tangent space $T_{(\psi,Y)}\Gamma$ to the graph γ at any point $(\psi, Y) \in \Gamma$ is given by

$$T_{(\psi,Y)}\Gamma = \left\{ \alpha\left(1, \frac{\partial F}{\partial \psi}(\psi, Y)\right) \in \mathbb{R}^2 \,\middle|\, \alpha \in \mathbb{R} \right\},$$

but $\frac{\partial F}{\partial \psi}(\psi, Y) = \frac{v_C}{b(\theta)}\sin(\psi_0 + \beta(\theta))$, and hence by (8b) and (9) we have $(\dot{\psi}, \dot{Y}) \in T_{(\psi,Y)}\Gamma$. $\square$

To change the lane, we change the state (Y, ψ) from $(y_0, 0)$ to $(y_1, 0)$. Without loss of generality, it is assumed that $y_0 > y_1$.

6.2.1 Manoeuvre with Constant Velocity

If we suppose that the velocity v_C during the entire manoeuvrer is kept constant, then suppose that $(\theta_0, \theta_1) \in [\underline{\theta}, 0) \times (0, \overline{\theta}]$ are such that the equation $F_{\theta_0,y_0,0,v_C}(\psi) = F_{\theta_1,y_1,0,v_C}(\psi)$, or equivalently

$$\tilde{y}_0(y_0, 0) - \tilde{y}_0(y_1, 0) + v_c\left(\frac{\cos(\psi + \beta(\theta_1))}{b(\theta_1)} - \frac{\cos(\psi + \beta(\theta_0))}{b(\theta_0)}\right) = 0,$$

has the solution $\hat{\psi}$. The proposed manoeuvre consists of

1. turning the front wheels from 0 to the angle $\theta_0 > 0$,
2. waiting until the orientation angle ψ is $\hat{\psi}$,
3. turning the wheels to the angle $\theta_1 < 0$,
4. waiting until the orientation angle ψ reaches 0,
5. finally turning the front wheels back to 0.

6.2.2 Manoeuvre with Varying Velocity of the Vehicle

Suppose the vehicle velocity v_C is piecewise constant on possibly very short time intervals. Let $\theta^*(t)$ be the solution of the following equation

$$F^*(\theta^*(t)) \equiv F_{\theta^*(t),Y(t),\psi(t),v_C(t)}(0) = y_1.$$

Notice that $\theta^*(t)$ depends on the current velocity $v_C(t)$.
Then the lane-change manoeuvre consists of

1. turning the front wheel from 0 to the angle θ_0,

2. waiting until the yaw angle $\psi(t)$ reaches ψ^* for some $\psi^* \in]0, \pi/2[$,
3. keeping the wheels at the angle $\theta(t) = \theta^*(t)$ until the orientation of the vehicle reaches 0 yaw angle.
4. turning the front wheels back to 0.

Both proposed controllers are feed-forward, thus a linear control [37] is to be implemented to remove deviations from the lateral reference y_1. The time t_{lc} of the manoeuvre depends on the vehicle velocity, v_C, and it is used in the guard of the abstract controller LCP depicted in Fig. 2.

7 Related Work

In the following, we consider related work within the categories of verification, hierarchical design approaches, spatial logics, and traffic maneuvers.

Automatic Verification. Most approaches to the automatic verification of hybrid systems represent discrete control and continuous dynamics together in one formal model, e.g., a hybrid automaton [2] or a hybrid program [36]. Whereas the reachability of locations is decidable for timed automata [3], this is in general not true for hybrid automata [18]. These limitations are overcome by using suitable abstractions and symbolic representations.

Model checking of linear hybrid automata by examining the reachable state space started with the tool HyTech [19]. More advanced techniques are incorporated in the tools PHAVer [12] and SpaceEx [13]. An alternative to these reachability-based methods are bounded versions of model checking using SAT-based techniques modulo the theory of ordinary differential equations [10, 11]. The concept of local theory extensions has been applied to proving safety properties of hybrid systems in [6]. Interactive theorem proving for hybrid systems in the context of an extended dynamic logic is pursued in [36]. For Hybrid CSP an experimental tool was developed [49].

Hierarchical Design. To simplify the analysis of hybrid systems, several approaches to controller design for hybrid systems have pursued a *separation* of the dynamics from the control layer.

An early work with an example of keeping distance between vehicles, is the paper by Nadjm-Tehrani and Strömberg [34], where they study the mapping from the continuous state space to the discrete state space. In the approach, the two models are combined to a hybrid model, and the linkage from the modes of the continuous model to the discrete modes is done by a *Characterizer* that generates events and a *Selector* for set-points. These could be characterized by linking predicates as done in this chapter, that would allow a clearer separation of the models.

Raisch et al. [31, 32] introduce abstraction and refinement to support a hierarchical design of hybrid control systems. However, this line of work stays within the same underlying model. Instead, the work here operates with separate models, because they can be tailored to the reasoning tools available for respectively automata and logics and those available for conventional control theory. Here, we are more in accordance with the work in [38], that deals with semantic alignment of heterogeneous models. The linking predicates introduced in the current work may make the alignment easier, because it relates only specific quantities and not full models.

Another inspiration for our work has been the approach pursued by Van Schuppen et al. [17] that works upwards from what we call the concrete model and introduce synthesis of control laws for piecewise-affine hybrid systems based on simplices, resulting in a discrete controller with transitions between the simplices. This may be an approach to finding a symbolic state space, when there is no obvious way to partition it.

Spatial Logic. Work on spatial logic often focusses on qualitative spatial reasoning [43] as exemplified in the region connection calculus [39]. We have used the spatial logic MLSL [21] to reason abstractly on highway traffic. The logic gives a compact formulation of properties and configurations, and an ability to compose and decompose them as well as a potential for deductions [26]. MLSL is inspired by interval temporal logic [33], the Duration Calculus [50], and the Shape Calculus [40]. It is a two-dimensional extension of interval temporal logic, where one dimension has a continuous space (the position in each lane) and the other has a discrete space (the number of the lane).

In [41], hybrid automata are considered where invariants and guards are expressed in a spatio-temporal logic $S4_u$. However, there is no separation of space and dynamics as in our approach.

Traffic Maneuvers. A very influential effort was the California PATH (Partners for Advanced Transit and Highways) project on automated highway systems for cars driving in groups called platoons [44]. The manoeuvres include joining and leaving the platoon, and lane change. Lygeros et al. [28] sketch a safety proof for car platoons taking car dynamics into account, but admitting *safe collisions*, i.e., collisions at a low speed. Not all scenarios of multi-lane traffic are covered in their proof.

Platzer et al. [5, 27] represent traffic applications in a differential dynamic logic $d\mathscr{L}$ that is supported by the theorem prover KeYmaera [36]. This logic does not separate space (symbolic model) from dynamics (concrete model), that is at the heart of our approach. The paper [1] proposes a bottom-up strategy, where a concrete model is gradually abstracted to Markov chains, for which the set of reachable states is analysed.

On highways, the analysis of safety is simplified because all cars drive in one direction. More difficult to analyse are country roads with opposing traffic. The safety of overtaking manoeuvres on such roads has been proven in [21]. Even more degrees of freedom in traffic manoeuvres can be found in urban traffic. The manoeuvres at crossings has been studied in [45].

Since driving assistants are liable to hit the road very soon, the effort at providing clear modelling and verification for this application area is very important.

Linking. For linking abstract and concrete data-manipulating systems the concepts of data and operation refinement with corresponding simulation-based proof techniques are well-known [8, 9]. Note that these techniques start by relating abstract and concrete data *variables*, that is not quite suitable in our setting, where we have to relate abstract *predicates* on reservations and claims to concrete sensor values. The transfer of temporal properties from abstract to concrete transitions systems via simulations and bisimulations is well-understood in the area of model checking [16].

8 Conclusion

This chapter has presented an approach to hybrid systems modelling where an abstract model is built in theories that are decidable modulo symbolic guards and actions while a concrete model uses conventional continuous time for which controllers are developed. The key point is that these two worlds are connected by *linking predicates*, so the concrete model is a refinement of the abstract one.

In the following, we discuss pros and cons of the approach for the individual steps and for the overall work.

Symbolic Model. A symbolic model is well known from a controller side, which can be built using timed automata. Also the use of symbolic guards and actions is intuitively easy. Note that time should enter only as timeouts on communications. These timeouts occur at the interface to the lower level concrete model or in communication protocols for interaction between the state machines.

When this is done, it is feasible to use model checking with a simplified environment model that assigns suitable values from finite domains to the predicates, and accept actions of similar finite types. Thus, an exhaustive automated verification is possible, although it has not been done in this chapter, because we consider the decomposition and linking the main points. Also, encoding the spatial model is a major effort. Steps in this direction have been taken by S. Linker in formalising a safety proof for a controller specification of [25] using the theorem prover ISABELLE.

Defining a suitable state space is intrinsically difficult. We have used a spatial logic to structure it. The logic gives a compact formulation of properties and configurations, and an ability to compose and decompose them as well as a potential for deductions. However, if a developer is not familiar with logic, it may be easier to stay with set theory, i.e., use the semantics underlying the logic. This would also be the case if a model checking tool is used, because the logic would have to be semantically encoded in most cases. The simple CTL or LTL logics used in model checkers are not nearly as expressive as spatial logics. Thus, the logic is not essential for the approach or even the application case, but it is a neat shorthand.

Concrete Model. Identification of the concrete model and controller development is well known and is highly application dependent. In the current presentation, the modelling and controller design is very general. For real applications there is much engineering to do, but this is not relevant for this exposition.

During the development, one must have an eye on the predicates of the symbolic model, so it is feasible to construct observers that match the guards, and handle set points presented by the actions.

Linkage. The linking predicates are the formal outcome of elaborate discussions concerning the interface of the two models. They represent the point where many real application projects fail, because engineering traditions from software development and control system development meet. The advantage of the approach is that the two sides have to meet and agree. An issue that is common to top-down approaches is that the defined interface turns out to be either unimplementable in the concrete or inadequate for the abstract verification. Here, we see no magic bullet.

Overall Comments. The approach seems well suited for application areas, where a collection of semi-autonomous entities have to coordinate to achieve common objectives. In a tightly coupled application, where there is a tight centralized supervisor, it is most likely easier to stay with a one level concrete model, typically a conventional hybrid automaton.

Acknowledgements We thank three anonymous reviewers for their helpful comments.

References

1. Althoff, M., Stursberg, O., Buss, M.: Safety assessment of autonomous cars using verification techniques. In: American Control Conference (ACC) 2007, pp. 4154–4159. IEEE (2007)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995)
3. Alur, R., Dill, D.L.: A theory of timed automata. TCS **126**(2), 183–235 (1994)
4. Ames, A.D., Cousineau, E.A., Powell, M.J.: Dynamically stable bipedal robotic walking with nao via human-inspired hybrid zero dynamics. In: HSCC 2012, pp. 135–144. ACM (2012)
5. Arechiga, N., Loos, S.M., Platzer, A., Krogh, B.H.: Using theorem provers to guarantee closed-loop system properties. In: American Control Conference (ACC) 2012, pp. 3573–3580. IEEE (2012)
6. Damm, W., Ihlemann, C., Sofroni-Stokkermans, V.: PTIME parametric verification of safety properties for reasonable linear hybrid systems. Math. Comput. Sci. **5**(4), 469–497 (2011)
7. Damm, W., Möhlmann, E., Rakow, A.: Component based design of hybrid systems: a case study on concurrency and coupling. In: HSCC 2014, pp. 145–150. ACM (2014)
8. de Roever, W.-P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press, New York (1998)
9. Derrick, J., Boiten, E.A.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Springer, London (2014)
10. Eggers, A., Fränzle, M., Herde, C.: SAT modulo ODE: a direct SAT approach to hybrid systems. In: Cha, S.D., Choi, J., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 171–185. Springer, Heidelberg (2008)
11. Fränzle, M., Herde, C.: HySAT: an efficient proof engine for bounded model checking of hybrid systems. Form. Methods Syst. Des. **30**(3), 179–198 (2007)
12. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. STTT **10**(3), 263–279 (2008)
13. Frehse, G., Guernic, C., Donzé, A., Cotton, S., Dang, T., Maler, O.: SpaceEx: scalable verification of hybrid systems. CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
14. Frehse, G., Kateja, R., Guernic, C.L.: Flowpipe approximation and clustering in space-time. HSCC **2014**, 203–212 (2013)
15. Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds).: Hybrid Systems. LNCS, vol. 736, Springer, Heidelberg (1993)

16. Grumberg, O.: Abstraction and reduction in model checking. In: Schwichtenberg, H., Steinbrüggen, R. (eds.) Proof and System-Reliabilty. Nato Science Series II. Math., Physics and Chemistry, vol. 62, pp. 213–260. Kluwer Academic Publishers, Boston (2002)
17. Habets, L., Collins, P., van Schuppen, J.: Reachability and control synthesis for piecewise-affine hybrid systems on simplices. IEEE Trans. Autom. Control **51**(6), 938–948 (2006)
18. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292. IEEE (1996)
19. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: a model checker for hybrid systems. STTT **1**(1–2), 110–122 (1997)
20. Hereid, A., Kolathaya, S., Jones, M.S., Van Why, J., Hurst, J.W., Ames, A.D.: Dynamic Multidomain Bipedal Walking with Atrias Through Slip Based Human-Inspired Control. HSCC 2014. pp. 263–272, ACM (2014)
21. Hilscher, M., Linker, S., Olderog, E.-R.: Proving safety of traffic manoeuvres on country roads. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 196–212. Springer, Heidelberg (2013)
22. Hilscher, M., Linker, S., Olderog, E.-R., Ravn, A.P.: An abstract model for proving safety of multi-lane traffic manoeuvres. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 404–419. Springer, Heidelberg (2011)
23. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall, London (1998)
24. Lee, E.A., Zheng, H.: Operational semantics of hybrid systems. HSCC 2005, 25–53 (2005)
25. Linker, S.: Proofs for traffic safety: combining diagrams and logic. Ph.D thesis, Dept. of. Comp. Sci, Univ. of Oldenburg (2015)
26. Linker, S., Hilscher, M.: Proof theory of a multi-lane spatial logic. Logical Methods Comput. Sci. **11**(3), 2015. See: https://arxiv.org/abs/1504.06986
27. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: hybrid, distributed, and now formally verified. In: Butler, M.J., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 42–56. Springer, Heidelberg (2011)
28. Lygeros, J., Godbole, D.N., Sastry, S.S.: Verified hybrid controllers for automated vehicles. IEEE Trans. Autom. Control **43**(4), 522–539 (1998)
29. Lynch, N.A., Segala, R., Vaandrager, F.W.: Hybrid I/O automata revisited. HSCC 2001, 403–417 (2001)
30. MathWorks. Stateflow (1995)
31. Moor, T., Raisch, J., Davoren, J.: Admissiblity criteria for a hierarchical design of hybrid systems. In: Proceedings IFAD Conference on Analysis and Design of Hybrid Systems, pp. 389–394. St. Malo, France (2003)
32. Moor, T., Raisch, J., O'Young, S.: Discrete supervisory control of hybrid systems based on l-complete approximations. Discret. Event Dyn. Syst. **12**, 83–107 (2002)
33. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. Computer **18**(2), 10–19 (1985)
34. Nadjm-Tehrani, S., Strömberg, J.: From physical modelling to compositional models of hybrid systems. In: Langmaack, H., de Roever, W.P., Vytopil, J. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS, vol. 863 of LNCS, pp. 583–604. Springer (1994)
35. Olderog, E.-R., Ravn, A., Wisniewski, R.: Linking spatial and dynamic models for traffic maneuvers. In: 54th IEEE Conference on Decision and Control (CDC), 8 pp. IEEE (2015)
36. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Spinger, Heidelberg (2010)
37. Rajamani, R.: Vehicle Dynamics and Control. Mechanical engineering series. Springer Science, New York (2006)
38. Rajhans, A., Krogh, B.H.: Compositional heterogeneous abstraction. In: HSCC 2013, pp. 253–262. ACM (2013)
39. Randell, D.A., Cui, Z., Cohn, A.G.: A spatial logic based on regions and connection. In: Proceedings 3rd International Conference Knowledge Representation and Reasoning (1992)

40. Schäfer, A.: A calculus for shapes in time and space. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 463–478. Springer, Heidelberg (2005)
41. Shao, Z., Liu, J.: Spatio-temporal hybrid automata for cyber-physical systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 337–354. Springer, Heidelberg (2005)
42. Sreenath, K., Hill Jr., C.R., Kumar, V.: A partially observable hybrid system model for bipedal locomotion for adapting to terrain variations. In: HSCC 2013, pp. 137–142. ACM (2013)
43. van Benthem, J., Bezhanishvili, G.: Modal logics of space. In: Aiello, M., Pratt-Hartmann, I., Benthem, J. (eds.) Handbook of Spatial Logics, pp. 217–298. Springer, Netherlands (2007)
44. Varaija, P.: Smart cars on smart roads: problems of control. IEEE Trans. Autom. Control AC **38**(2), 195–207 (1993)
45. Werling, M., Gindele, T., Jagszent, D., Gröll, L.: A robust algorithm for handling traffic in urban scenarios. In: Proceedings of IEEE Intelligent Vehicles Symposium, pp. 168–173. Eindhoven, NL (2008)
46. Woodcock, J., Davies, J.: Using Z – Specification, Refinement, and Proof. Prentice Hall, New Jersey (1996)
47. Zabat, M., Stabile, N., Farascaroli, S., Browand, F.: The aerodynamic performance of platoons: a final report. UC Berkeley (1995). http://escholarship.org/uc/item/8ph187fw
48. Zabczyk, J.: Mathematical Control Theory – An Introduction. Birkhäuser (2008)
49. Zhan, N., Wang, S., Zhao, H.: Formal modelling, analysis and verification of hybrid systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Unifying Theories of Programming and Formal Engineering Methods. LNCS, vol. 8050, pp. 207–281. Springer, Heidelberg (2013)
50. Zhou, C., Hoare, C., Ravn, A.: A calculus of durations. IPL **40**(5), 269–276 (1991)
51. Ziegler, J., Bender, P., Dang, T., Stiller, C.: Trajectory planning for bertha – A local, continuous method. In: 2014 IEEE Intelligent Vehicles Symposium Proceedings, Dearborn, MI, USA, June 8-11, 2014, pp. 450–457 (2014)

Towards Interface-Driven Design of Evolving Component-Based Architectures

Xin Chen and Zhiming Liu

Abstract The sustainable development of most economies and the quality of life of their citizens largely depend on the development and application of *evolutionary digital ecosystems*. The characteristic features of these systems are reflected in the so called *Internet of Things* (*IoT*), *Smart Cities* and *Cyber-Physical Systems* (*CPS*). Compared to the challenges in ICT applications that the ProCoS project used to face 25 years ago, we today deal with systems with the complexity of ever evolving architectures of networked digital components, physical components, together with sensors and devices controlled and coordinated by software. The architectural components, also called subsystems, are designed with different technologies, run on different platforms and interact through different communication technologies. However, the ProCoS project goal remains valid and the critical requirements of applications of these systems should not be compromised, and thus critical components need to be "*provably correct*". This chapter is in a form of a summary and position paper to discuss how software design for complex evolving systems can be supported by an extension of interface-driven rCOS method that we have recently been developing. We show the need for an *interface theory* to underpin development of techniques and tools. We demonstrate the need of multi-modelling notations for the description of multi-viewpoints of designs to help mastering system complexity, and their theoretical foundation in the nature of Unifying Theories of Programming proposed by Sir Professor Tony Hoare and Professor He Jifeng, as part of the outcome of the ProCoS project.

Zhiming Liu—The work is funded by the project SWU 116007, and China NSF Grant 61672435.

X. Chen
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
e-mail: chenxin@nju.edu.cn

Z. Liu (✉)
Centre for Software Research and Innovation, Southwest University, 2 Tiansheng Rd, Beibei, Chongqing 400715, China
e-mail: zhimingliu88@swu.edu.cn

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_6

1 Introduction

In the post-industry era, the challenges of the global concern of sustainable development depend on innovation application *digital ecosystems*. Such a system exists in the form of a distributed network of smart devices, program controlled physical systems (such as machines in future manufacturing factories and devices in hospitals), digital computing systems and services on the Web (or clouds). The digital components and physical objects with embedded electronics, software and sensors, which interact and collaborate through different communication networks and protocols. Such a system is open and evolving from both of

1. the key feature of the system that allows to plug-and-play new system components and services, and allows legacy components to be adapted, upgraded or replaced, and
2. the key feature of the business, social and knowledge communities it supports that are ever changing and growing.

The generally known Internet of Things (IoT) [26], Smart Cities [35] and Cyber-Physical Systems [20] are different forms of digital ecosystems. They are becoming major networks of infrastructures for development of applications in all economic and social areas such as healthcare, environment management, transport, enterprises, manufacturing, agriculture, governance, culture, societies and home automation. These applications share a common model of architectures and involve different communication technologies and protocols among the architectural components. The research and applications thus require collaborations among experts with expertise in a variety of disciplines and various skills in software systems development.

The openness of the architecture, heterogeneity of components and the scale (or complexity) of both functionality and interactions impose challenges beyond the capacity of the state of the art of software engineering. One of the most fundamental problems is that either the traditional top–down or the bottom–up development strategy, or any combination of both kinds, cannot be readily used to the development and maintenance of digital ecosystems. Therefore, there exist no methods and tools to support systematic development of digital ecosystems and their front-end applications. Ad-hoc development using tailored existing methods and tools is far from meeting the following essential requirements:

- safe and secure integration of new digital and cyber-physical components;
- maintenance and healthy evolution of legacy components and services;
- consistent adaptation of existing Internet and cloud services and applications to new and special-purpose services/devices;
- development of new applications and services from existing services/devices;
- data collection from different sources with different components, interoperably communicating among different components for processing, analytics and support of decision making.

To advance beyond the state of the art of software engineering, we need a model that captures the ever-evolving nature of the system architectures, allowing dynamically integration and replacement of different devices, services and components. We need to develop software engineering techniques and their tool support for

1. incrementally building the model of the evolving architecture,
2. interface-based development of new components and front end applications, and their integration into an existing architecture,
3. interface-based adaption and reuse of legacy components in an existing architecture, and
4. validation and verification of components and systems by using integrated tools of simulation, testing and formal verification of trustworthiness (safety, security, privacy and dependability).

The architectural model should also support the design of fault-tolerance [10, 27, 36] with techniques of runtime monitoring and recovery [17]. Simulation with large amount of data is also needed in building models, where the data are either known or collected in the model building process, say through sensors.

In what follows, we discuss, in Sect. 2, the characteristic of complexity of digital ecosystems to clarify the challenges stated above and to give a background motivation to the interface-driven approach to health system evolution. In Sect. 3, we introduce the basics of the rCOS formal model-driven method of component and object system. We give an example in Sect. 4 to show how rCOS supports incremental and interface-driven design. In Sect. 5, we propose an extension of rCOS to modelling cyber-physical component systems.

2 Complex Evolving Systems

Software engineering was born with the aim to deal with the inherent complexity of software development, and its vision was that complexity should be mastered through the use of models, techniques and tools developed based on the types of theoretical foundations and practical disciplines that have been established in the traditional branches of engineering [28, 33]. The directions and contents of software engineering and their advances are defined and driven by the following fundamental attributes of software complexity [1–3]:

1. the complexity of the domain application,
2. the difficulty of managing the development process,
3. the flexibility possible to offer through software, and
4. the problem of characterising the behaviour of software systems.

The first attribute is the source of the challenges in software requirements *gathering, specification, analysis* and *validation*, that are the main topics of *software requirements engineering*. The second attribute, driving the development of *software project management*, concerns the difficulty to define and manage a development process to deal with complex and changing requirements of software projects that involve a large team of software engineers and domain experts. The process has to identify the software technologies and tools that support collaboration of the team in working on shared software artifacts. The third attribute concerns the difficulties in the design of software architecture, and the design and reuse of software components, algorithms and platforms. The final attribute of software complexity pinpoints the challenges in modelling, analysis, validation and verification of the behaviour of the software.

2.1 Chronic Complexity of Digital Ecosystems

The fundamental attributes of software complexity are all reflected in software of digital ecosystems, but their extensions are becoming increasingly wider, due to the increasing power of these systems, here we quote

> "The major cause of the software crisis is that the machines have become several orders of magnitude more powerful. To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."
>
> — Edsger Dijkstra
>
> The Humble Programmer, Communications of the ACM [9]

Now not only we have gigantic computers, but also networked computers of all scales of power from micro devices, through systems with multi-cores and multiprocessing units, to supercomputers. They execute programs anywhere and any time, which share data and communicate and collaborate with each other. These digital ecosystems are represented by the popular Internet of Things (IoT), Smart Cities, Data Centres and Cyber-Physical Systems (CPS). There exist not much agreed or clear characteristic descriptions of these systems, and a variety of viewpoints and classification exist for them. In fact, it is reasonable not to distinguish them [11, 18], especially when we are interested in system modelling, design, verification and validation. They all share the following attributes of these complex and evolving systems

1. They bring together computation, physical objects and processes, electronics, and networking communication to seamless integration of and close interaction between the physical world and computer-based systems.
2. The actions of these systems, as well as the objects, are monitored, coordinated, controlled and integrated by computing systems and existing network infrastructures.
3. These system are constantly evolving, such that new digital systems, embedded devices and physical processes keep being integrated into the system, and legacy digital systems, devices and physical processes keep being removed, modified and reconfigured.

We consider systems with the above characteristics which have *component-based* or *system of systems architectures*. Some researchers intend to distinguish component-based systems from systems of systems and say that the latter have *emergent behaviour*. We interchange these two terms as there is no clear definition on what emergent behaviour of CPS is. Complex evolving systems exhibit the following features that are the causes of major challenges in their modelling, analysis and design:

1. Different components of these systems can have different data models, such as patients' records in healthcare systems. This feature implies the requirements of interoperable communication and information sharing.
2. Such a system has multi-stakeholders and multi-endusers who have different viewpoints of the system and whose applications use different computing, data and network and physical resources and services of the system.
3. The composition and coordination of distributed computations and services also support collaborative workflows involving multi-users.
4. Diversity of requirements of safety, security, privacy, timing, and fault-tolerance.

2.2 An Application Examples

The example as shown in Fig. 1, is a smart grid, taken from the presentation at an UK Innovate event [31]. Such a system includes smart metering and advanced metering infrastructure that provides intriguing opportunities to embrace new sustainable services for the whole energy value chain [8, 38].

A network of smart meters can also be part of the grid to provide real-time pricing for all types of users and so encourage individual consumers to reduce their power consumption at peak times. To this end, consumers can adjust their own individual

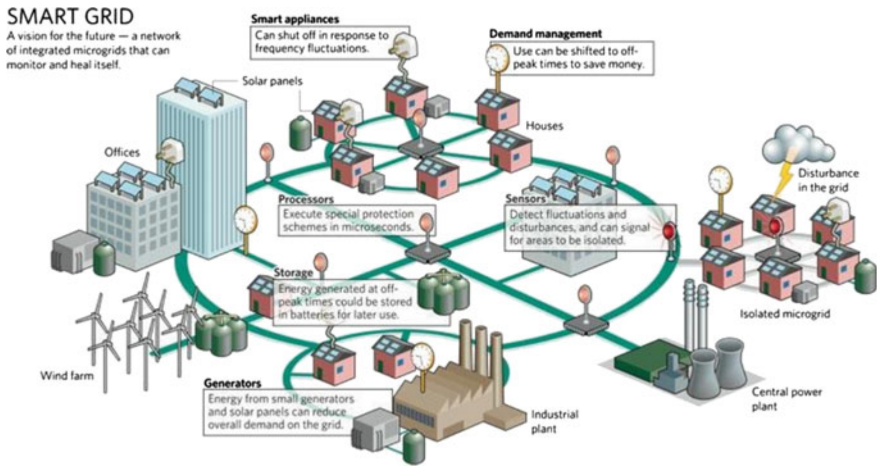

SMART GRID
A vision for the future — a network of integrated microgrids that can monitor and heal itself.
Smart appliances
Can shut off in response to frequency fluctuations.
Demand management
Use can be shifted to off-peak times to save money.
Solar panels
Offices
Houses
Disturbance in the grid
Processors
Execute special protection schemes in microseconds.
Sensors
Detect fluctuations and disturbances, and can signal for areas to be isolated.
Storage
Energy generated at off-peak times could be stored in batteries for later use.
Isolated microgrid
Wind farm
Generators
Energy from small generators and solar panels can reduce overall demand on the grid.
Industrial plant
Central power plant

Fig. 1 Smart grid

load according to the time-differentiated prices. Furthermore, smart meters, software and communication together also enable consumers to cooperate aiming at achieving energy-aware consumption patterns, in order to realise for example, the demand-side management, demand response and Direct Load Control programmes. For illustration, imagine a smart community that autonomously adapts its energy consumption by means of enabling a limited number of household smart meters to share real-time neighbourhood information cooperatively. Users therefore cooperate with each other and with data collectors, thus facilitating the integration of energy consumption information into a common view. We will propose to develop a model of an evolving network of smart meters in Sect. 5. As in branches of transitional engineering, handling the above challenges involves the best practice of the fundamental principles of *separation of concerns*, *divide and conquer*, and *use of abstraction* through information hiding (in different design stages).

3 Interfaces and Component-Based Architectures

We now introduce the model of component interfaces that we have developed in the rCOS methods - Refinement of Component and Object Systems. The work on the rCOS framework includes formal semantics of an OO specifications, an OO refinement calculus, a unified model of component-based and OO model, that are available in a number of publications, e.g. [5, 7, 12, 13]. This chapter provides a summary and linkages among these models and theories without going into formal details. We have also published work about a UML profile for rCOS and tool support to model constructions and transformations based on the profile [21, 23, 25]. Therefore, the UML diagrams used in this chapter all have formal semantics in rCOS.

The rCOS method intends to support model-driven design (MDD) of complex evolving system. This is characterised by *letting system design be carried out in a process through building system models* to gain confidence in requirements and designs. The process of model construction in MDD emphasises on

- the use of *abstraction* for information hiding so as to be well-focused and problem oriented;
- the use of the engineering principles of *decomposition* and *separation of concerns* for *divide and conquer* and *incremental development* and evolution; and
- the use of *formalisation* to make the *process repeatable* and *artefacts (models) analysable*.

3.1 Key Features of rCOS

Main differences of the rCOS method from other model-based formal frameworks, such as Circus [4, 29], are rather in philosophic principles and intentions, instead of

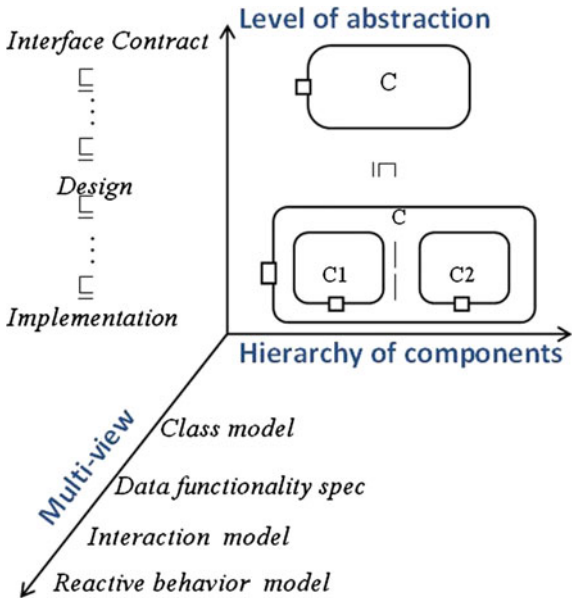


Fig. 2 rCOS modelling approach

expressive power. For example, the rCOS method makes components and interfaces as first class modelling concepts and elements, and explicitly and systematically supports separation of concerns with its multi-dimensional modelling approach to component-based architecture modelling, as shown in Fig. 2.

- First, it allows models of a component at different levels of abstraction, from the top level models of *interface contracts* of components, through models produced at different design stages including *platform independent models* (PIM) and *platform specific models* (PSM), to models of deployment and implementations.
- At each level of abstraction, a component has models of different viewpoints, including the *class model* (or *data model*), the specification of *static data functionality* (i.e. changes of data states), the *model of interaction protocol* with the environment (i.e. actors) of the components, and the *model of reactive behaviour*. These models of different viewpoints support the understanding of different aspects of the components and support different techniques of analysis, design and verification of different kinds of properties.
- A model of a component is hierarchical and composed from models of 'smaller' components that interact and collaborate with each other through their interfaces. Some components can also control, monitor or coordinate other components.

The significant advantage is that it allows the model of a component or a system at a level of abstraction is synthesised from the models of the data model, functionality and architecture, while these individual models can be refined in separation to preserve their consistency. More distinguished features of rCOS include

- direct object-oriented abstraction, instead of coding classes, objects and polymorphism in process-oriented models with unstructured states [6, 13];
- fully supported by a sound and complete object-oriented refinement calculus [37];
- direct formulation of OO design patterns as refinement rules [32, 37];
- provision of model transformations from component-based models of architecture of requirements to OO models of design architecture [6], and from OO models of design architectures to component-based models of design architectures [21];
- the provision of a well defined UML profile so that models can be constructed using the subset of UML defined by the profile and automatically translated to into rCOS models [24, 25].

The feature in the last bullet point allows us to use UML to represent models in the rest of the chapter.

3.2 Components and Their Interfaces

Components are service providers - including computing devices realising functions, processes that coordinate and control components through interactions and connectors. We intend to have different types of interfaces for different interaction mechanisms and protocols. Here, we only use a running example to show the rCOS modelling notation and method.

To ease the understanding and practice, we divide the definition of component into its syntactic description and semantic specification that we call the *contract* of the component. A (**syntactic**) **component** is represented by tuple $C = \langle X, IF, A \rangle$, where

- X is a finite set (possibly empty) of state variables.
- IF is the (**provided**) **interface** defining a finite set of operation signatures of the form $m(\overline{x}; \overline{y})$ with a finite number of input parameters and a finite number of return parameters. Each operation represents service provided to users.
- A is a finite set (possibly empty) of **internal actions**, each of which is represented as a parameterless method a. An internal action is automatous and does not have parameters.

For example, a memory can be modelled as component that provide a write operation and read operation to its user, e.g. a processor.

```
Component M
    Z d;
    provided interface MIF {
        W(Z v); R(; Z v)
    }
```

A faulty memory can be modelled as a component below, that provide write and read operations to the user (e.g. a processor) but its content can be corrupted by an internal 'fault action'.

```
component $fM$
    $Z\ d$;
    provided interface $MIF$ {
        $W(Z\ v)$; $R(;\ Z\ v)$;
    }
    actions{//fault modelling corruption
        $fault$
    }
```

The syntactic interface defines the static type of the component, but it does not specify the behaviour of the interface. The behaviour of an interface is specified by a contract. For incremental understanding, we first define a *service contract* of an interface, which specifies the state change of an execution of interface operation, provided, required or internal operations.

A **service contract** $\mathcal{C}$ of a syntactic component C specifies

- an **initial condition** defining the allowed possible initial states of the variables X by a state predicate $\mathcal{C}.init$ on X, called the **initial condition**;
- a **state transition relation** $\mathcal{C}.next$ that specifies each operation $m(\overline{x};\overline{y})$ in the provided interface IF by a pair $P \vdash R$ of a precondition P and a postondition R, where
  - P is a predicate over $X \cup \overline{x}$,
  - R is a predicate over $X \cup \overline{x} \cup X' \cup \overline{y'}$, and X' and $\overline{y'}$ are the sets of the primed version of the variables in X and $\overline{y}$.

  The meaning of $P \vdash R$ is that from a state s of X with the input parameters $\overline{x}$ satisfying precondition P, the execution of $m()$ will change the state s of X into a state s' (in which the value of x is represented by x') with the return values $\overline{y}'$ such that $((s,\overline{x}),(s',\overline{y}'))$ holds for R.
- the **state transition relation** $\mathcal{C}.iNext$ that specifies each internal operation a in A by pair $P \vdash R$ of a precondition P and a postondition R, where
  - P is a predicate over $X \cup \overline{x}$,
  - R is a predicate over $X \cup X'$.

In general, it is proven in UTP [16] that all programming statements in traditional structured programming languages can be defined by designs. In particular, an assignment $x := e$ is defined as design $\{x\} : true \vdash x' = e$, meaning that the state is changed from a state s to a new state s' in which only the value of x is changed to the evaluation of e in s, keeping other variables unchanged. The following specification combines the syntax and the service contract of a memory component offering the environment a write operation and a read operation.

```
Component $M$
    $Z\ d$;
    provided interface $MIF$ {
        $W(Z\ v)\{d := v\}$; $R(;\ Z\ v)\{v := d\}$
    }
```

The design calculus in UTP [16] is extended to object-oriented designs in [13, 37].

A service contract only specifies the functionalities of the component in terms of a *contract* between the *assumption* on the current state and input parameters and the *guarantee* on the change of the state and return values. However, a component is in general *reactive*, thus also controls its interaction protocol with the environment and the dependency (or causality) relation between its operations. The flow of control and interaction are specified by the *guards* of the operations:

- the **guard** of an operation $m()$ in the interface IF or the international action set A is a predicate on X such that $m()$ can be executed in a state only when its guard holds in the state and the action is disabled in the state otherwise.[1]

Thus, a (guarded) contract $\mathcal{C}$ of a component actually defines a *labeled state transition system*, but the states combine both control and data together, and the labels are the interface operations and internal operations. $\mathcal{C}$ specifies each operation $m()$ by a triple of a guard g, a precondition P and a post condition R, denoted by $g\&(P \vdash R)$, called a **guarded design**. A transition from a state s to a state s' by an operation $m()$, provided, required or internal, is possible only if its guard, denoted by $\mathcal{C}.guard(m)$, holds in s. And when it is possible

- if the precondition P of $m()$ holds in s, then R holds for the pair (s, s') of states together with relation between the input and return parameters if $m()$ is a provided or required interface operation; and
- if the precondition P of $m()$ does not hold in s, s' can be any state.

When we separate the control states from the data states in the state transition system of $\mathcal{C}$, we obtain an automaton with the control states and the interface signatures as the alphabet. This allows us to use the language defined by the automaton, a regular expression when the automaton is of finite states, to express the interaction protocols.

We propose a textual specification of components in a format similar to Java, that allows us to declare multiple interfaces. In the corresponding abstract definition of components, the provided interface IF is the union of the declared interfaces. We take a few simple examples to illustrate the concepts of components. For example, the following reactive component specifies a memory that controls the order in which the write and read operations are invoked.

```
Component B
    $Z\ d,\ Bool\ w = 1;$
    provided Interface $BIF$ {
        $W(Z\ v)\{w\&\{d, w\} : true \vdash d' = v \wedge w' = \neg w\};$
        $R(;\ Z\ r)\{\neg w\&\{v, w\} : true \vdash v' = d \wedge w' = \neg w\}$
      }
    }
```

This memory also behaves like a one-place buffer.

[1]In general, the guard can contain input parameters, and even the primed version y' of return parameters y in $\overline{y}$, especially when advanced security assurance is required. We do not consider this general case as we have no semantics yet to handle them.

3.3 Composition and Orchestration

We can easily see that the one-place buffer B can be built by coordinating the uncontrolled memory M

```
Component $B$ requires $M$
    $Bool\ w = 1;$
    provided Interface $BIF$ {
        $W(Z\ v)\{w\&(M.W(z);\ w := 0)\};$
        $R(;\ Z\ r)\{\neg w\&(M.R(;\ r);\ w := 1)\}$
      }
    }
```

We use regular expression to specify the protocol of control, obtaining the following equivalent specification

```
Component $B$ requires $M$
    provided Interface $BIF$ {
        $W(Z\ v)\{M.W(z)\};$
        $R(;\ Z\ r)\{M.R(;\ r)\}$
        protocol $\{(WR)^* + (WR^*)W\}$
      }
    }
```

Thus, a coordination mainly changes the interaction protocol of a component, such as M, without changing the data functionality of the component. Later in Sect. 3.4, we will see a visual model the protocol can be represented as state machine diagram in the rCOS UML profile [23].

With given components, we can construct new components with connectors and through orchestration of the provided operations in the given components. For example, taking $B_i = B[W_i/W, R_i/W]$ is obtained by the connector that renames the write $W()$ and read $R()$ operations of B to $W_i()$ and read $R_i()$, respectively, for $i = 1, 2$, we can have

```
component $M_2$ requires $B_1, B_2$ {
    $Z\ y;$
    provided interface $M_2IF$ {
        $move()\{B_1.R_1(;\ y);\ B_2.W_1(y)\};$
    }
```

This component provides the newly added $move()$ and the operations that B_1 and B_2 provide minus those that are called in the body of $move()$. And the protocol is defined by the guard conditions of B_1 and B_2. In general, we can extend a given set of components to form new components by defining additional provided operations using structured programming constructs. we can also use the *internalising* connector to make a provided operation, such as $move()$, internal. for example $Buff2 = M_2\backslash move()$ behaves as

```
component Buff2 requires $B_1$, $B_2$
    $Z\ y$;
    actions $A$ {
        $move()\{B_1.R_1(;\ y);\ B_2.W_1(y)\}$;
    }
```

Component *Buff2* behaves like M_2, except for *move()* will be executed internally and autonomously when it is enabled, without the need to be called from the environment. Thus, it behaves like a two-place buffer.

Now we give an specification of the faulty memory, in which an interaction protocol is specified using an regular expression that can be coded as guards of the interface operations.

```
component fM
    $Z\ d$;
    provided interface MIF {
        $W(Z\ v)\ \{d := v\};\ R(;\ Z\ v)\ \{v := d\}$;
    protocol $\{(WR)^* + (WR)^*W$// protocol of $C\}$
    }
    actions{//fault modelling corruption
        $fault\ \{true| - d' <> d\}$
    }
```

We use the renaming operators on the (provided) interface of fM and obtain three faulty memory components $fM_i \mathrel{\widehat{=}} fM[fM_i.W/W,\ fM_i.R/R]$, for $i = 1, 2, 3$. We now specify the following component.

```
component V requires $fM_1, fM_2, fM_3$ {
    provided interface VIF{
        $W(Z\ v)\ \{fM1.W(v); fM2.W(v); fM3.W(v)\}$;
        $R(;\ Z\ v)\ \{v := vote(fM1.R(v), fM2.R(v), fM3.R(v))\}$;
        protocol $\{(WR)^* + (WR)^*W\}$
    }
```

We can prove the proposition that *the composition of* V *is refinement of the perfect component* $B = C||M$ *if it is assumed at any time at most one of the* fM_i *is in faulty state* [27, 36]. The component-based architecture is shown in Fig. 3.

3.4 Separation of Concerns

When the data model for the variables, interface interaction protocols and the dynamic behaviour of component become complex, models of different viewpoints for different design concerns are needed. To this end, we have a UML profile for rCOS [24]. This allows that for object-oriented design of component-based modelling and design of finite state components, we use

- UML *class models* for the representation of the data models at different levels of abstraction, specially *conceptual class model* for requirements and *design class models* for object-oriented design of components;

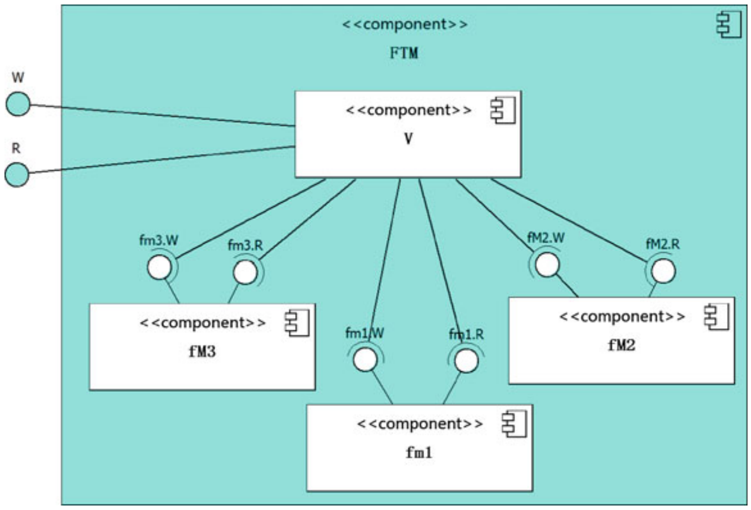


Fig. 3 Component-based architecture of a fault-tolerant memory

- (extended) UML *sequence diagrams* for modelling interactions among components and between components and actors (component sequence diagrams), and for interaction among objects of a design of a component (*object sequence diagrams*); and
- (extended) UML state machine diagrams for modelling the dynamic behaviour of a component.

The extended sequence diagrams, together with the textual specification of pre- and post-conditions of the methods, generate the rCOS functionality definitions of the participating components, such as V, and the state diagrams of the components define the protocols that are corresponding to the guards of the methods in the components. Thus, the contracts of the interfaces can be divided into the **contracts of static functionality** and the **contracts of dynamic behaviour**. The former are given by the unguarded design of interface operations that are specified only by their pre- and post-conditions, and the latter by the state machine diagram of the components. With a UML profile defined for rCOS, these models of different views points can be automatically integrated into rCOS textual specification [23, 25].

The sequence diagrams and state machine diagrams of different viewpoints of f the fault-tolerant memory are shown in Fig. 4, and we will discuss more examples in the next section.

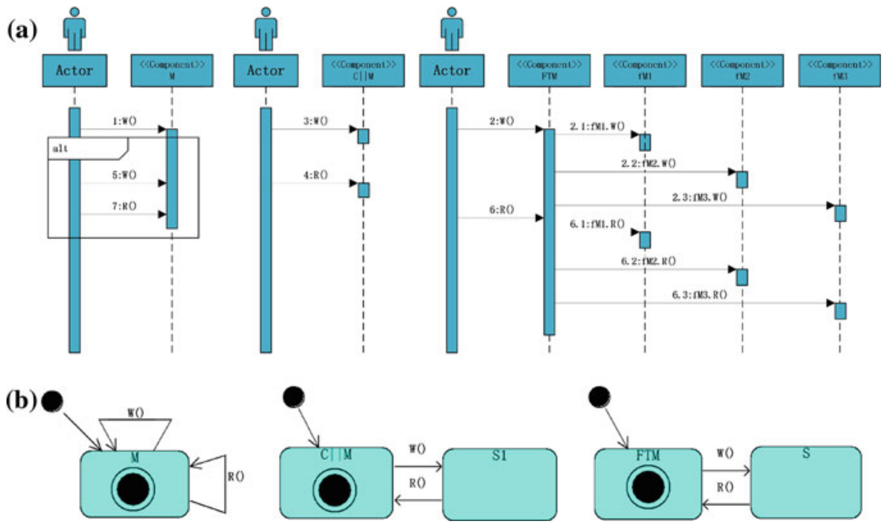


Fig. 4 UML models of interactions and dynamic behaviour

4 Incremental Design of an Enterprise Application

Incremental/evolutionary modelling and design has been practised in empirical and ad hoc software development. This section, however, demonstrates how rCOS supports an incremental/evolutionary modelling and design of the case study of a computerised trading system of an enterprise of supermarkets. It was used as the Common Component Modelling Example (CoCoME) [6, 14]. It is an extension of the Point of Sale (POST) example used in Larman's textbook [19]. The case study was described in terms of the use cases related to *process sales*, *manage inventory*, *prepare for product orders*, *process deliveries of ordered products*, and *exchange products among different stores*, etc.

The evolutionary nature of the system is determined by the development of the enterprise. The business may just start from a single store and the store requires a computerised system to improve the automation of the use case process sales to speed up customer checkout and record the sales. Also, at the early stage of the business, only one checkout "cash desk" is enough, or the system development can start with considering only one checkout cash desk.

4.1 Requirements Modelling

The requirements gathering and analysis starts from describing use cases, and any described use case explicitly or implicitly implies restrictions on the functionality either due to the stage of the business development or consideration for a simplification to start with. For example, we start with the use case *process sale with cash payment* briefly described below.

Overview: A customer arrives at the Cash Desk with the product items to purchase with cash payment. The sale and the payment are recorded in the system. Involved Actors includes Customer and Cashier.

Process: The normal courses of interactions between the actors and the system are described as follows.

1. When a Customer comes to the Cash Desk with her items, the Cashier initiates a new sale. The system creates a new sale.
2. The Cashier enters each item, either by scanning in the bar code or by some other means; if there is more than one of the same item, the Cashier can enter the quantity. The system records each item and its quantity and calculates the subtotal.
3. When there are no more items, the Cashier indicates to the system the end of entry. The total of the sale is calculated. The Cashier tells the Customer the total and asks her to pay.
4. The Customer gives the Casher cash and the Cashier enters the amount received. The system records the cash payment amount and calculates the change. Then the completed sale is logged.

Alternative courses of events: There are exceptional or alternative courses of interactions, e.g., if the entered bar code is not known in the system, the Customer does not have enough money for a cash payment. A system needs to provide means of handling these exceptional cases, such as cancel the sale.

At the requirements stage, *we model a use case as a component* by a **conceptual class model**, a **component sequence diagram**, **state machine diagram**, and the **contract of static functionality** of the interface operations. For the use case process sale with cash payment, we have the class model in Fig. 5, sequence diagram in Fig. 6a, and state machine diagram in Fig. 6b.

The operations that actor Cashier calls in Fig. 6a form the provided interface of component *ProcessSale*, and the state machine diagram in Fig. 6b defines its contracts of dynamic behaviour. Their consistency can be checked by FDR [34] after being translated into processes of CSP [15, 34]. The contract of static functionality of *ProcessSale* is specified by the pre- and post-conditions of the interface operations.

The precondition of *startSale*() requires the existence of the *Store*, the *CashDesk*, the *Catalog* and the *Product Specifications*. The postcondition of *startSale*() is to create a new sale. Thus, the state variables of *ProcessSale* include *Store store*, *CashDesk cashdesk*, *Catalog cat*, and *Sale sale*. The contract of *startSale*() can be specified as

$$\{store \neq nil \land cashdesk \neq nil \land cat \neq nil\}\ startSale()\ \{sale' = new\ Sale\}$$

Similarly, we can specify the contracts of the other operations. For example,

$$\begin{bmatrix} \wedge store \neq nil \\ \wedge cashdesk \neq nil \\ \wedge cat \neq nil \\ \wedge sale.isComplete \end{bmatrix} makeCashPayment(a) \begin{Bmatrix} \wedge cashPay'{=}new\ CashPayment \\ \wedge cashPay'.amount = a \\ \wedge Is\text{-}Pay\text{-}by(sale, cashPay') \end{Bmatrix}$$

The semantics of OO contracts of operations are derived from OO designs in [13].

4.2 OO Design of Components

In practical but informal OO development, the design stage is to decompose the functionality and responsibility of each interface operation (informally) described by it pre- and post-conditions and assign the sub-responsibilities to "appropriate" objects of the component. The decomposition and assignment of the responsibilities are carried out using GRASP design patterns [19]. These patterns are proven to be rCOS refinement rules [13, 37]. Therefore, the following design steps can actually be formally justified in rCOS.

For a requirements model of a component, such as that of *ProcessSale* given in the previous subsection, we design each interface operation according to its contract. This is done by using the formalised GRASP design patterns and refactoring rules that are formally proven in the OO refinement calculus [37]. In particular, by Controller pattern, we can decide to implement the provided interface of *ProcessSale* by class

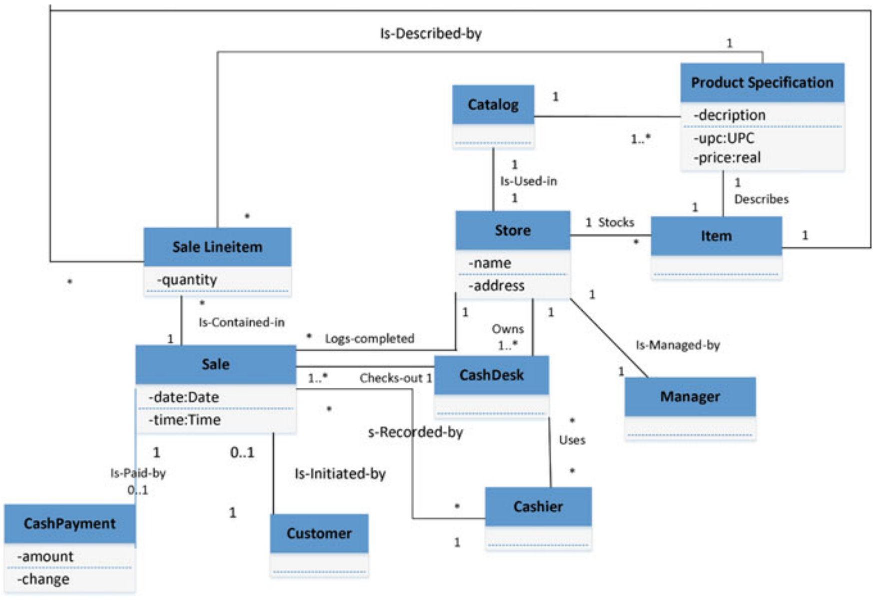


Fig. 5 Conceptual class diagram of *ProcessSale*

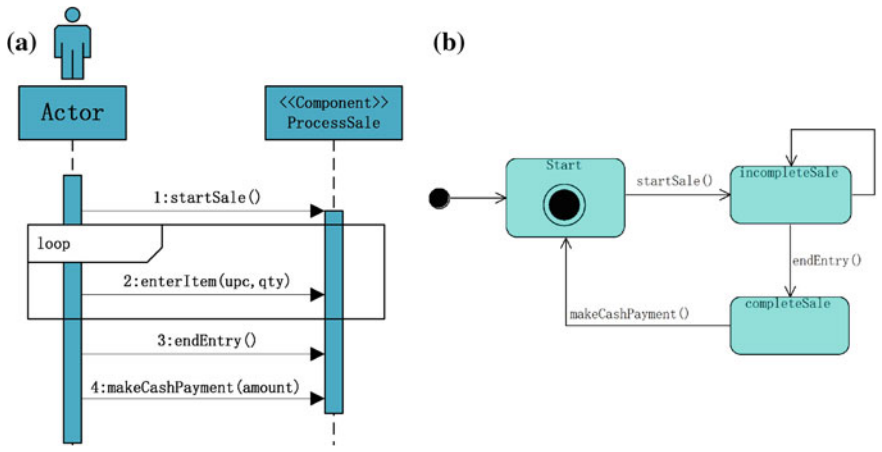


Fig. 6 Sequence diagram and state machine diagram of *ProcessSale*

CashDesk, and the design of each operation is represented by an **object sequence diagram**. For example, the design of *makeCashPayment*() is given in Fig. 7.

With the model transformation tool of rCOS [22], we can check that the objects : *CashDesk*, *sale: Sale* and *:CashPayment* form a component *HandleSale* with the interface object: *CashDesk*; and the objects *:Store* and the container object ⟨⟨*Set*⟩⟩*Sale* form another component *StoreManagement* with the interface object : *Store*. The tool then automatically transforms the OO design in Fig. 7 to a component-based design in Fig. 8b.

The design proceeds with OO design of the other provided interface operations of *ProcessSale*, followed by decomposition into provided interface operations, *startSale*(), *enterItem*(*upc*, *qty*), and *endEntry*() of *HandleSale*, and required interface operations of *HandleSale* for *checking* the validity of *upc*, and *extracting* the product specification from the *Catalog* object continued in the *Store* object. Therefore, the *upc* checking operation *check*(*upc*) and specification *extracting* operation *find*(*upc*; *spec*) are provided interface *ManageStore*. We then obtain a component-based decomposition of

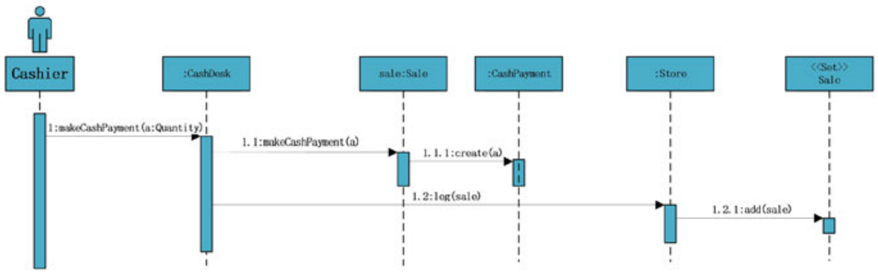


Fig. 7 OO design of *makeCashPayment*()

component *ProcessSale* shown in Fig. 8a. The rCOS transformation tool [22] also automatically generates the *static component diagrams* shown in Fig. 9 corresponding to the transformation from the OO design in Fig. 7 to the component-based design in Fig. 8.

4.3 Incremental Development and System Evolution

Component *ProcessSale* designed in the previous subsection assumes some restrictions on the functionality. For example, among other restrictions, it deals cash payment only and has no inventory update when the completed sale is logged. In general, in each cycle of Rational Unified Development Process, components and their individual operations are designed for restricted functionalities. Further development is to relax the assumptions to extend their functionalities, and to design new components. The rCOS method also put such incremental and evolutionary design into its formal refinement calculus so as to ensure rigorous correctness. We informally show such incremental design by singling out the process of handling cash payment as a use case by itself, denoted by *HandleCashPayment*.

We take the operations represented by messages 1–3 in Fig. 6a to form a component, denoted by *HandleSale*. Component *HandleCashPayment* itself can be designed as a component with the provided interface operation *makeCashPayment*(). Its OO design is the same as that in Fig. 7, and the component-based decomposition is the same as that in Fig. 8b, but with a new component name *HandleCashPayment*. In a new development cycle, we can follow the same way in which component *HandleCashPayment* is modelled to design a model of component *HandleCreditPayment*. It provides an operation *makeCreditPayment*(). Before a *CreditPayment* is created, *HandleCreditPayment*

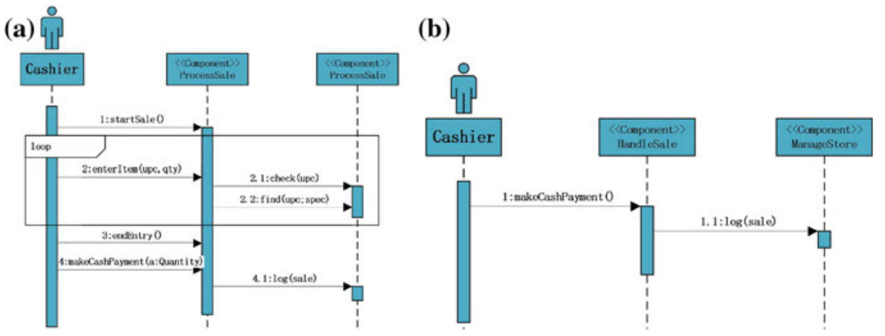


Fig. 8 Component sequence diagram of component *ProcessSale*

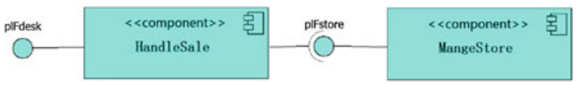


Fig. 9 A component diagram

calls the service from actor *Bank* for the authorisation of the credit payment. Therefore, *HandleCashPayment* requires to call an operation of the Bank, that we denote by $authoriseCredit(cardInfo, amount)$. After authorisation, the *CreditPayment* is created, and the completed sale is logged to *Store*. In the same way, we design a component *HandleCheckPayment*.

Assume that a system that only supports process sale with cash payment is already developed. In its system evolution, a new component *HandleCreditPayment* can be specified through investigation of the original architecture that consists *HandleSale* and *ManageStore*. This new component can then be designed and integrated into the legacy architecture to support processing credit payment.

With the architecture models in Figs. 8 and 9, we can extend the provided interface of *ManageStore* with more product management operations, such as those for changing the price of a product, increasing and deducting the inventory of a product (after more items are ordered and sold). We can then upgrade component *HandleProcessSale* so that after the complete sale is logged to the *Store*, the product items of the sale are removed from stock using the inventory deduction operation, say $deInventory(upc, qty)$. This can be realised by *aspect oriented* design and the interface operation *makeCashPayment* (and $makeCreditPayment()$) first executes its original body and then calls the method $decInventory(cpu, qty)$ of *ManageStore* repeatedly for each item in the sale. This is an "*after*" advice in aspect oriented design. An aspect oriented architecture modification like this is modelled as a connector component that changes the original component by modifying the execution of the interface operation according to the advices in the aspect.

Further system evolution can go from one checkout cash desk to a number of them in a store, from an one-store business to an enterprise of a store chain. Also, further extension to the system can be developed to support online shopping. The model of component-based architecture and interfaces contracts are also imported for analysis of safety, security and performance vulnerabilities and deficiencies so that architecture modifications and changes of interaction protocols can be designed to improve the safety, security and performance.

5 Towards Modelling Cyber-Physical Component Systems

The components in the previous section are digital components. We now propose to extend the models to *physical interfaces* and *cyber-physical components*, using the evolutionary development of a smart meter network demand response (DR) programme [30].

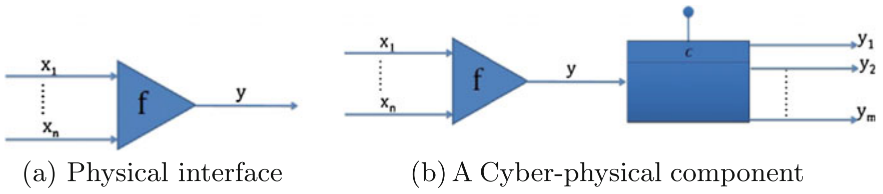


Fig. 10 Cyber-physical component

5.1 Physical Interfaces and Cyber-Physical Components

We extend the model of components with variables, called *physical variables*, whose behaviour are functions from time to real number, depending on conditions of digital states. The trajectories of the physical variables are specified by differential equations. For example, the rate of electricity consumption of an electrical appliance are different when the appliance is in different states, say when it is "on", "off", or in the "energy-saving" state. We model a *physical interface* as function $f(x_1, \ldots, x_n; y)$ with one or more *incoming signals* $x_1, \ldots, x_n$ that are continuous variables, and one[2] *outgoing signal* y, as shown in Fig. 10a. The incoming signals of an interface are also called *requiring signals*. A component also provides (or outputs) signals to the environment, such as $y_1, \ldots, y_n$ in cyber-physical component shown in Fig. 10b. There, the function f defined in the component is part of hybrid behaviour of the component, and the solid circle represents the provided *digital* (or cyber) interface. The definition of operations in the provided cyber operations may rely on operations to be provided by other operations, called the *required cyber interface* and represented by the half circle in the diagram. The composition of components is also extended by linking provided signals of a component to incoming signals of interfaces of another component.

5.2 Model the Evolution of a Smart Meter Network

The system in this case study consists of three kinds of components.

- **Consumer**: is a household equipped with one or more smart meters that is connected to the power line, electrical appliances, and to a communication network.
- **Data Collector**: is in charge of the data aggregation process. According to the resource allocation algorithm, this process is modelled as a centralised coordinator, but a distributed approach can be implemented securely.
- **Utility**: is a set of energy suppliers shared by customers. We assume utilities to implement distributed generation

[2]In general, there can be more than one.

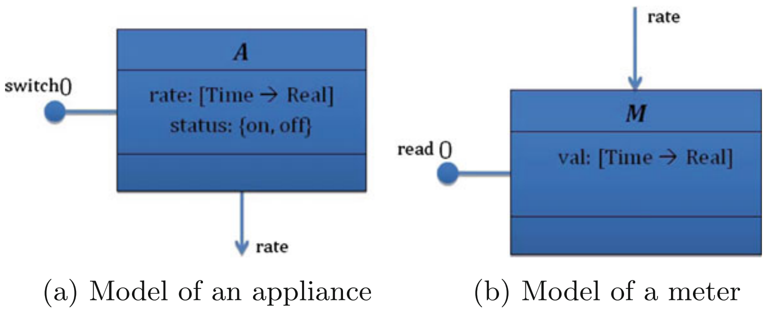


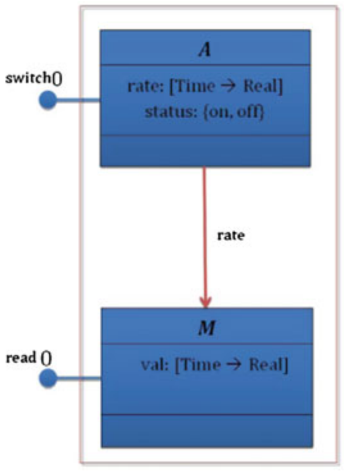
Fig. 11 Appliance and meter

We mainly demonstrate the evolutionary nature of the system and show how our modelling approach scales up. We first consider a single appliance A of a single household. An appliance, as shown in Fig. 11a, has a digital state *Status* which takes a value *on* or *off*, and it is changed by the *digital interface* operation *switch*(). The appliance has an observable signal *rate* representing the electricity consumption rate. It is a function from "Time" to real numbers, that (presumedly) can be obtained from manufacturer of the appliance. The signal "rate" is useless if the householder only observes the "rate" and switches on the appliance when needed.

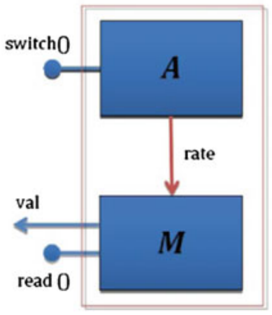
If the householder wants to know better about his daily use of electricity and to plan his use of the appliance in order to reduce their electricity bill, an electronic meter M can be introduced as shown in Fig. 11b. Meter M records the accumulated consumption of energy of an appliance A. Its provided interface *M.pIF* provides a digital operation *read*() and its required interface $M.rIF$ consists of a single signal *rate*. The interface behaviour of M (i.e., the return of *read*()) is a discretised value of the internal signal *val* that is a timed function dependent on the required signal rate. For example, it can be defined as $val(t) = \int_0^t ratedx$. In general, the the trajectories of the continuous variables of a component C are specified as timed functions of the form $\gamma C = F(\beta C, \gamma C, rW)$, where feedbacks loops are possible. If we compose the appliance A and the meter M, we have the component shown in Fig. 12a.

There are alternative models. For example, a meter can include a sensor that observes the *rate*. Then val would be discretised and represented as a step function. Then *read*() directly returns the value of the internal discrete variable *val*. In this model, the sensor is actually represented as part of the physical interface. Also, a meter can be modelled as a component with a required signal rate and a provided digital operation $val()$ to the meter component. The advantage of the component-based modelling with explicit interface contracts is exactly to allow different models and support comparative analysis.

At this stage of system evolution, the *read*() and *switch*() are still only manually operated by the householder. A further desire of home automation is to introduce a control component P, called a *control pad*. For accuracy and fault-tolerance, we make the internal signal *val* of $A\|M$ external, and denote this cyber-physical component as



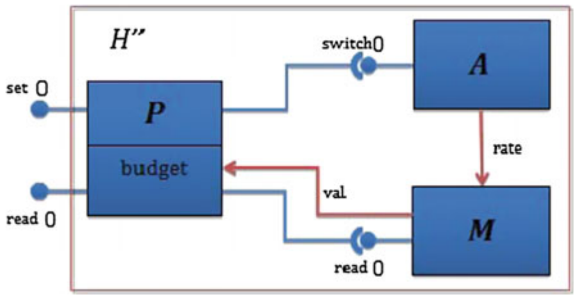
(a) Composition $A \| M$ of an appliance and a meter



(b) A cyber-physical model of the composition MA

Fig. 12 Models of composition of an appliance and a meter

Fig. 13 Automatically controlled appliance



MA, as shown in Fig. 12b. We now compose the control pad P and component MA, and it is shown in Fig. 13. Now the householder can use set and $read()$ to program daily use of the appliance, according to a daily budget.

Home automation The evolution continues and a household can have a number of appliances. Then more meters or a meter with an open number of required input signals can be used for the design of a control pad. The overall control pad can either be designed using the existing individual control pads or the individual control pads are replaced with a centralised control pad. In either case, the design models of the individual control pads can be reused. The advantage of the proposed framework is that a household with a number of appliance can be treated in the same ways as if the household has a single appliance.

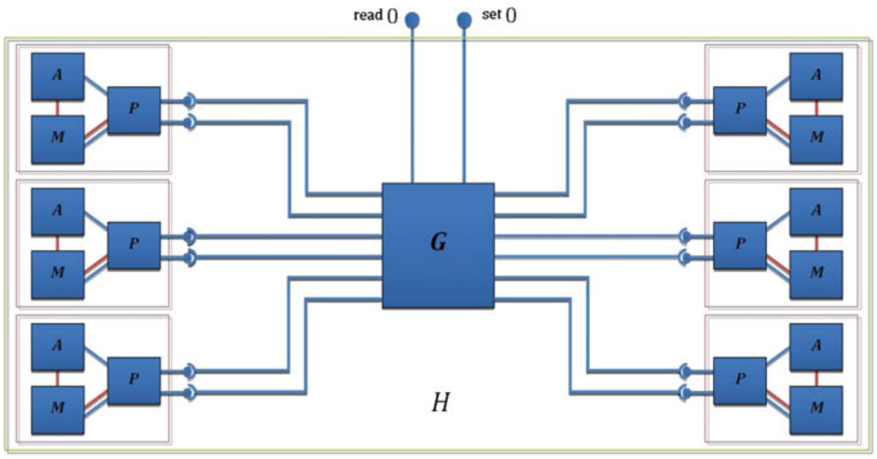
$$\begin{array}{ll}
A_i \widehat{=} A[switch_i/switch, rate_i/rate], & A \widehat{=} A_1 \parallel \ldots \parallel A_n \\
M_i \widehat{=} A[read_i/read, val_i/val], & M \widehat{=} M_1 \parallel \ldots \parallel M_n \\
P_i \widehat{=} [set_i/set, read_i/read, val_i/val, switch_i/switch], & P \widehat{=} P_1 \parallel \ldots \parallel P_n
\end{array}$$

Here, renaming of interface operations and signals are used. We add a global controller P for planning and schedule of a household, and thus obtain an automated household $H \mathrel{\widehat{=}} G\|P\|M\|A$. This is shown in Fig. 14a. This system is closed inside the house and thus there are no security threats to it (unless a burglary happens). However, a further step of evolution can introduce a controller operated through a mobile phone, as shown in Fig. 14. We denote this automated home by MH. Then, an open mobile phone communication network is used, and security threats are introduced too. Therefore, interface-driven component-based architectures are essential to identify system safety vulnerabilities, security threats, and performance deficiencies, so as to make architecture modifications to enhance safety, security, availability and fault-tolerance.
Network evolution The designs of a household can be abstractly described as follows.

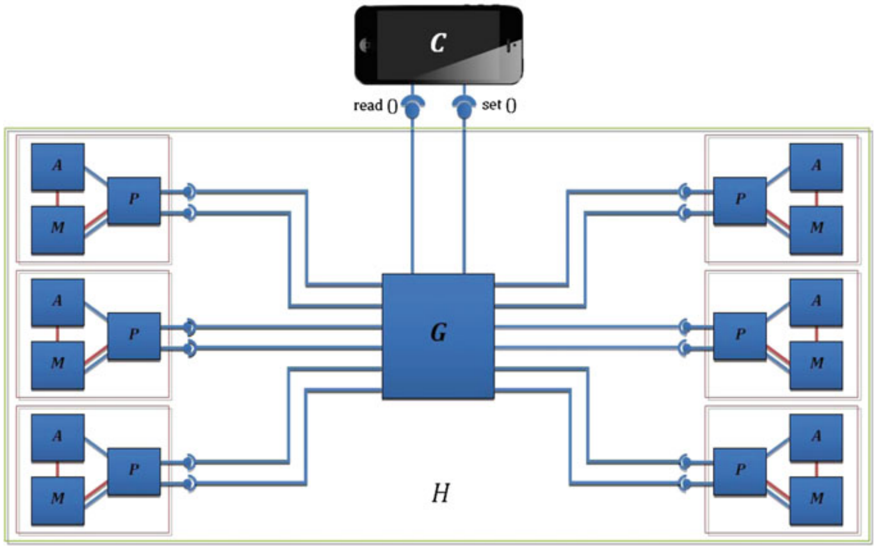
```
Component H {
    attributes: fD, vD: Real;//fixed and variable
                            //energy demands of the community
    signal: val: Real;
    provided interface:
         Rf(;x:Real), Rv(;y:Real);
         Wf(x:Real), Wv(y:Real);
         setUp() /** set up budget and policy /** by householder;
         val/ ** provided signal

     Functionality:
     Rf(;x){x:=fD}; Rv(; y){y:=vD};
     Wf(x){fD:=x}; Wv(y){vD:=y}
     ----
   }
```

We define a network of households $H \mathrel{\widehat{=}} H_1 \parallel \ldots \parallel H_k$ for a community of residence. Assume the component $Utility$ provides a operations $requestF(x{:}Real;\, u{:}\ Real)$ and $requestV(y{:}Real;\, v{:}\ Real)$ for supply of fixed energy and variable energy, respectively. When it is called, the method returns the amount of committed supply for the day through the return parameter. Consider a $Coordnator$ component which periodically calls the interface operations $Rf_i()$ and $Rv_i()$ and makes a request to $Utility$ through $request()$. After it receives notification from $Utility$ about the committed supply, it "negotiates" with the households (through communication interfaces that we omit in this chapter) and reallocates budgets to the households through $Wf_i()$ and $Wv_i()$. This gives a network system $H\|Coordinator\|Utility$, as shown in Fig. 15.

Except for the "negotiation" of the *Coordinator* with individual households, the composition H of the households behaves exactly the same as one household. Similarly, we can imagine that a network of utilities works in collaboration to provide a power supply. Once they reach an agreement among themselves on how they share the supply to the request from the collector, they interface with the collector in the



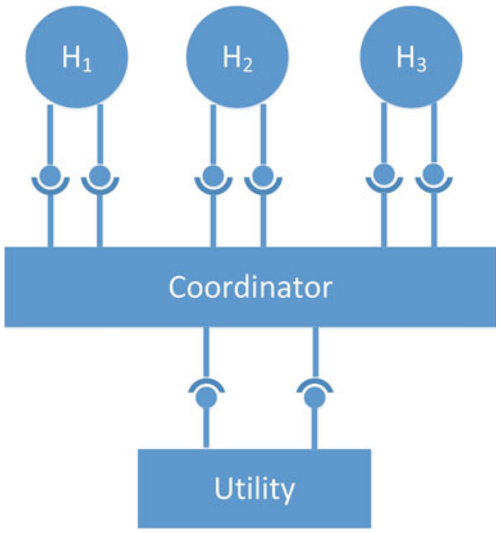
(a) Automated household



(b) A mobile controlled home

Fig. 14 Home automation

Fig. 15 A smart grid



same manner as a single utility. Furthermore, the centralised collector can be transformed into a distributed implementation so that the “negotiation” can be performed among households themselves.

6 Conclusions

This chapter has argued the importance of component-based (or system of systems) architectures and contracts of interfaces for healthy evolution of digital ecosystems. We proposed an extension to the rCOS model of digital components and interfaces to cyber-physical components. This makes the notion of interfaces very general. For example, a piece of wall or a window can be modelled interfaces between the temperatures outside and inside a room. Even the “air” between two sections of a room can modelled as an interface that transforms the temperature of one section to that of another. However, this general notion of interfaces poses a number of challenges, for example

1. How to develop a model of contracts of such interfaces, as it is often the case that there is no known physical laws or functions for defining these interfaces?
2. How to define the formal semantics and the refinement relation between cyber-physical interface contracts?

These are the first significant questions to ask when developing a semantic theory for these CPS components and their compositions. Further challenges include

1. how to develop design techniques and tools,

2. how to combine David Parnas's Four-Variable Model, Michael Jackson's Problem Frames Model, and the Rational Unified Process (RUP) of the use case driven approach systematically into the continuous evolutionary integration system development process?

We believe that our model-driven approach is again promising, and techniques and tools of simulation with rich data and machine learning would become increasingly important in building the correct models.

Acknowledgements We acknowledge the contribution to the development of the rCOS method from Zhenbang Chen, Ruzheng Dong, He Jifeng, Wei Ke, Dan Li, Xiaoshan Li, Jing Liu, Charles Morisset, Anders Ravn, Volker Stolz, Shuling Wang, Jing Yang, Liang Zhao, and Naijun Zhan. We also thank Jonathan Bowen, Xiaohong Chen, Sabita Maharjan, Esther Palomar and Yan Zhang for the collaboration on Component-Based Modelling for Sustainable and Scalable Smart Meter Networks [30].

References

1. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Boston (1994)
2. Brooks, F.P.: No silver bullet: essence and accidents of software engineering. IEEE Comput. **20**(4), 10–19 (1987)
3. Brooks, F.P.: The mythical man-month: after 20 years. IEEE Softw. **12**(5), 57–60 (1995)
4. Cavalcanti, A., Sampaio, A., Woodcock, J.: A refinement strategy for circus. Form. Asp. Comput. **15**(2–3), 146–181 (2003). http://dx.doi.org/10.1007/s00165-003-0006-5
5. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) International Symposium on Fundamentals of Software Engineering. Lecture Notes in Computer Science, vol. 4767, pp. 191–206. Springer, Berlin (2007)
6. Chen, Z., Hannousse, A.H., Hung, D.V., Knoll, I., Li, X., Liu, Y., Liu, Z., Nan, Q., Okika, J.C., Ravn, A.P., Stolz, V., Yang, L., Zhan, N.: Modelling with relational calculus of object and component systems–rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.) The Common Component Modeling Example. Lecture Notes in Computer Science, chap. 3, vol. 5153, pp. 116–145. Springer, Berlin (2008)
7. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Sci. Comput. Program. **74**(4), 168–196 (2009). Feb
8. Darby, S.: Smart metering: what potential for householder engagement? Build. Res. Inf. **38**(5), 442–457 (2010)
9. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972). An ACM Turing Award lecture
10. Fischer, C.: Fault-tolerant programming by transformations. Ph.D. thesis, University of Warwick (1991)
11. Gunes, V., Peter, S., Givargis, T., Vahid, F.: A survey on concepts, applications, and challenges in cyber-physical systems. Trans. Internet Inf. Syst. **8**(12), 4242–4268 (2014)
12. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. **160**, 173–195 (2006)
13. He, J., Liu, Z., Li, X.: rCOS: a refinement calculus of object systems. Theor. Comput. Sci. **365**(1–2), 109–142 (2006)
14. Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R., Krogmann, K., Koziolek, H., Mirandola, R., Hummel, B., Meisinger, M., Pfaller, C.: The common component modeling example. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common

Component Modeling Example. Lecture Notes in Computer Science, chap. 1, vol. 5153, pp. 16–53. Springer, Berlin (2008)
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
16. Hoare, A., He, J.: Unifying Theories of Programming. Prentice Hall, New York (1988)
17. Kim, M., Viswanathan, M., Lee, I., Ben-Abdellah, H., Kannan, S., Sokolsky, O.: Formally specified monitoring of temporal properties. In: Proceedings of the European Conference on Real-Time Systems (1999)
18. Koubaa, A., Andersson, B.: A vision of cyber-physical internet. In: Proceedings of the Workshop of Real-Time Networks (RTN 2009), Satellite Workshop of ECRTS 2009 (2009)
19. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice-Hall, Upper Saddle River (2001)
20. Lee, E.: Cyber physical systems: design challenges. Technical Report No. UCB/EECS-2008-8, University of California, Berkeley (2008)
21. Li, D., Li, X., Liu, Z., Stolz, V.: Interactive transformations from object-oriented models to component-based models. In: Arbab, F., Ölveczky, P.C. (eds.) Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14–16, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7253, pp. 97–114. Springer (2011). http://dx.doi.org/10.1007/978-3-642-35743-5_7
22. Li, D., Li, X., Liu, Z., Stolz, V.: Interactive transformations from object-oriented models to component-based models. In: Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14–16, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7253, pp. 97–114. Springer (2011)
23. Li, D., Li, X., Liu, Z., Stolz, V.: Support formal component-based development with UML profile. In: 22nd Australian Conference on Software Engineering (ASWEC 2013), 4–7 June 2013, Melbourne, Victoria, Australia. pp. 191–200 (2013)
24. Li, D., Li, X., Liu, Z., Stolz, V.: Support formal component-based development with UML profile. In: 22nd Australian Conference on Software Engineering (ASWEC 2013), 4–7 June 2013, Melbourne, Victoria, Australia. pp. 191–200. IEEE Computer Society (2013). http://dx.doi.org/10.1109/ASWEC.2013.31
25. Li, D., Li, X., Liu, Z., Stolz, V.: Automated transformations from UML behavior models to contracts. SCI. CHINA Inf. Sci. **57**(12), 1–17 (2014). http://dx.doi.org/10.1007/s11432-014-5159-8
26. Li, X., Lu, R., Liang, X., Shen, X., Chen, J., Lin, X.: Smart community: an internet of things application. Commun. Mag. **49**(11), 68–75 (2011)
27. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. ACM Trans. Program. Lang. Syst. **21**(1), 46–89 (1999)
28. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969)
29. Oliveira, M., Cavalcanti, A., Woodcock, J.: Formal development of industrial-scale systems in *Circus*. ISSE **1**(2), 125–146 (2005). http://dx.doi.org/10.1007/s11334-005-0014-0
30. Palomar, E., Liu, Z., Bowen, J.P., Zhang, Y., Maharjan, S.: Component-based modelling for sustainable and scalable smart meter networks. In: Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM 2014, Sydney, Australia, June 19, 2014. pp. 1–6 (2014)
31. Pronios, N.B.: Software verification & validation for complex systems, presentation at Technical Feasibility Studies Competition Information Event, Innovate UK
32. Quan, L., Qiu, Z., Liu, Z.: Formal use of design patterns and refactoring. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13–15, 2008. Proceedings. Communications in Computer and Information Science, vol. 17, pp. 323–338. Springer (2008). http://dx.doi.org/10.1007/978-3-540-88479-8_23

33. Randell, B., Buxton, J. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27–31 Oct. 1969, Brussels, Scientific Affairs Division, NATO (1969)
34. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (1997)
35. Shapiro, M.: Smart cities: quality of life, productivity, and the growth effects of human capital. Rev. Econ. Stat. **88**, 324–335 (2006). May
36. Zhang, M., Liu, Z., Morisset, C., Ravn, A.P.: Design and verification of fault-tolerant components. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. Lecture Notes in Computer Science, vol. 5454, pp. 57–84. Springer, Berlin (2009)
37. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects Comput. **21**(1–2), 103–131 (2009). Feb
38. Zhu, J., Pecen, R.: A novel automatic utility data collection system using ieee 802.15.4-compliant wireless mesh networks. In: Proceedings of IAJCIJME International Conference (2008)

Part V
Automatic Verification

Computing Verified Machine Address Bounds During Symbolic Exploration of Code

J Strother Moore

Abstract When operational semantics is used as the basis for mechanized verification of machine code programs it is often necessary for the theorem prover to determine whether one expression denoting a machine address is unequal to another. For example, this problem arises when trying to determine whether a read at the address given by expression a is affected by an earlier write at the address given by b. If it can be determined that a and b are definitely unequal, the write does not affect the read. Such address expressions are typically composed of "machine arithmetic function symbols" such as `+`, `*`, `mod`, `ash`, `logand`, `logxor`, etc., as well as numeric constants and values read from other addresses. In this chapter we present an abstract interpreter for machine address expressions that attempts to produce a bounded natural number interval guaranteed to contain the value of the expression. The interpreter has been proved correct by the ACL2 theorem prover and is one of several key technologies used to do fast symbolic execution of machine code programs with respect to a formal operational semantics. We discuss the interpreter, what has been proved about it by ACL2, and how it is used in symbolic reasoning about machine code.

1 Preface

One might ask why a chapter on the ACL2 project is included in the volume marking the 20th and 25th anniversaries of the European ProCoS project. ProCoS was in part inspired by the successful effort at Computational Logic, Inc. (CLI), first published in 1989, to verify a system "stack," from a gate-level description of a microprocessor, through an assembler, linker, loader, two compilers, and an operating system, to several applications. All were verified using the Nqthm [5] theorem prover and their correctness results were designed to compose so that each level relieved the preconditions of the level below. The result was a mechanically checked theorem of the form: under certain very specific preconditions on the resources available and the

J Strother Moore (✉)
Department of Computer Science, University of Texas, Austin, TX, USA
e-mail: moore@cs.utexas.edu

inputs, the application programs (when compiled, linked, and loaded) run correctly on the hardware. The only unverified assumptions were the ones at the bottom: the fabrication of the gate-level description was faithful to the design and the physical gates behave as logically specified [1].

But the CLI stack inspired more than ProCoS. It was one of several Nqthm projects in the late 1980s and early 1990s involving models of commercial interest. See for example the work on the C String Library as compiled by `gcc` for the Motorola 68020 [6]. These projects stressed Nqthm in ways we had not seen before: its capacity, efficiency, and convenience as a practical functional programming language. Thus was born, in 1989, ACL2: A Computational Logic for Applicative Common Lisp [7, 11–13] ACL2 was a reimplementation of Nqthm in an applicative subset of Common Lisp [19]. But while the logic of Nqthm was a "homegrown" dialect of pure Lisp, the logic of ACL2 is applicative Common Lisp, a fast, efficient, widely supported ANSI standard programming language.

ACL2 has since been used in many industrial projects and is in use regularly at several companies involved with microprocessor design. For a good illustration of how ACL2 can be used in industry, see [18].

2 Introduction

Operational semantics has long been used to formalize and mechanically verify properties of machine code programs. Examples of the Edinburgh Pure Lisp Theorem Prover, Nqthm and ACL2 being used to prove functional correctness of code under formal operational semantics may be found in numerous publications [1, 2, 6, 10, 16, 17, 20, 21].

In such applications, terms in the logic are used to represent machine states, transition functions define the effects of individual instructions, these instruction-specific transition functions are then wrapped up into a "big switch" single-step function that applies the transition function dictated by the opcode of the next instruction, and finally the single-step function is wrapped up into a recursive iterated step function for giving semantics to whole programs. Typically the program being analyzed is stored in the state, either encoded numerically in memory or symbolically in some "execute only" state component. Theorems are then posed, typically, as implications asserting that if the initial state has some property then the "final" state produced by the iterated step function has some related property. These theorems are typically proved by induction but the "heavy lifting" in the proof is done by a rewriting strategy that explores the various paths through the program and composes and simplifies the individual state transitions. The rewriting strategy is just deductive implementation of *symbolic evaluation* which we sometimes also call *code walking*. The basic idea of symbolic evaluation is to start with a symbolic state expression containing a concrete program counter and program code but containing variables in some state components (e.g., memory locations holding program data). Hypotheses typically constrain these variables. To symbolically step that state: retrieve the instruction at the program counter, instantiate the transition function with that instruction, simplify the resulting state (rearranging expressions representing the contents of various registers and memory locations, testing them, and producing an `IF`-expression with new states with known program counters), and repeat on all the new states until some condition is satisfied.[1]

We sometimes refer to these proofs as *code proofs* because they can establish properties of explicit machine code.

Fundamental to this approach to semantics are the terms denoting reads and writes to the memory of a state because every transition requires manipulating the memory. In this work we focus on a byte addressed memory and use these terms for read and write:

$\texttt{R}(a, n, st)$:	returns the natural number obtained by reading n bytes starting at address a in the memory of state st
$\texttt{!R}(a, n, v, st)$:	returns the new state obtained by writing n bytes of natural number v into the memory of st starting at address a

We call a an *address* and n an *extent*. `R` and `!R` use the Little Endian convention to represent natural numbers as sequences of bytes. If an integer is supplied for v above, its twos complement representation – a natural number – is used. For example, `!R` writes the least significant byte of the binary representation of v into address a and writes the more significant bytes into the higher addresses.

`R` and `!R` enjoy certain properties that are crucial to code proofs. One such property is:

$$a, n, b, m \in \mathbb{Z} \wedge (a + n \leq b) \rightarrow \texttt{R}(a, n, \texttt{!R}(b, m, v, st)) = \texttt{R}(a, n, st).$$

Such a theorem is called a *read-over-write* theorem because it tells us about the results of reading after writing. This particular read-over-write theorem says the write can be ignored if the read fetches bytes in memory addresses below those written. There are other theorems to deal with overlapping reads and writes and reads above writes. There are analogous *write-over-write* theorems for simplifying state expressions. All are crucial to code proofs.[2]

[1]The process just described is just ordinary mathematical simplification of the iterated step function applied to the initial state. A special case of symbolic evaluation is “symbolic simulation” or “bit blasting” by which we mean a process whereby objects from a given finite set are represented using nested structures whose leaves are Boolean constants and variables. The process computes related objects from definitions or other equations using Boolean decision methods typically based on binary decision diagrams (BDDs) or Boolean satisfiability procedures (SAT). ACL2 supports symbolic simulation, e.g., see the ACL2 online documentation topic GL, but in this chapter we are concerned with straightforward simplification.

[2]Typical machine state models involve many other state components, their “accessor” and “updater” function symbols, and their analogues to “read-over-write” theorems, etc. But we ignore them in this chapter since we are focused on address resolution.

But what is of concern here is how, in a theorem proving context, we establish such inequalities as $(a + n \leq b)$ when a, n, and b are given by terms produced by symbolic evaluation of machine code. Such hypotheses litter the read-over-write and write-over-write conditional rewrite rules that are heavily used in code proofs. These rules are typically tried many more times than they are successfully applied: given an arbitrary read-over-write expression one must try to establish the hypotheses of each rule to determine whether the read is below, overlapping, or above the write. Furthermore, in typical code proofs, thousands of read-over-write expressions are encountered. Finally, the expressions a and b can become very large.

To put some numbers on the adjectives "heavily used," "large," etc., consider the largest symbolic state encountered while symbolically exploring a machine code implementation of the DES algorithm. The state in question represents the end of one path through the 5,280 instructions in the decryption loop. The normalized state expression contains 2,158,895 function calls, including 58 calls of `!R` to distinct locations and 459,848 calls of `R`. (Repeated writes to the same location are eliminated by the rewriting process.) That state expression also contains 1,698,987 calls of arithmetic/logical functions such as addition, subtraction, multiplication, modulo, and bitwise logical `AND`, exclusive `OR`, shift, etc. The largest value expression written is given by a term involving 147,233 function applications, 31,361 of which are calls of `R` and the rest are calls of arithmetic/logical functions. Values written often become indices into arrays and thus become part of address expressions.

We found it impractical to use ACL2's conventional arithmetic library to answer the address comparison questions that arise while building up such large state expressions. But ACL2 allows the user to extend the rewriter with special-purpose symbolic manipulation programs if those programs — which are written in the ACL2 programming language — are first proved correct by ACL2. So we developed special-purpose programs to answer such questions as "is $(a + n \leq b)$ true?" or more generally, "how do the values of expressions a and b compare?" The core technology is an Abstract Interpreter over Natural Number Intervals called `Ainni`, which takes a term and the context in which it occurs and tries to compute a bounded natural number interval containing all possible values of the term in that context. `Ainni` is purely syntactic — it just walks through the term bounding every subterm — and can be thought of as a verified type-inference mechanism where the types are intervals. `Ainni` was then used to develop a variety of metafunctions for manipulating the gigantic expressions produced by the symbolic evaluation of machine code sequences containing thousands of instructions.

In Sect. 3 we give some practical information about ACL2 as well as explain ACL2 notation which we often use in place of conventional notation because our techniques involve metafunctions which manipulate the internal ACL2 representation of terms. In Sect. 4 we discuss that representation and metafunctions. In Sect. 5 we introduce ACL2's pre-existing notion of "bounder" functions and a library of elementary bounders. In Sect. 6 we describe the key idea: `Ainni`, our abstract interpreter for machine arithmetic expressions that attempts to produce a bounded interval containing the value of the expression. Also in this section we show the correctness results for `Ainni`. These results have been proved by ACL2 and are necessary if `Ainni` is to be used in verified metafunctions. In Sect. 7 we illustrate calls of `Ainni` and the interpretation of its results. In Sect. 8 we exhibit a metafunction that uses `Ainni` to simplify a certain kind of `MOD` expression. This section shows how a metafunction assembles the results of `Ainni` into a provably correct answer. In Sect. 9 we briefly describe other applications of `Ainni`, including the motivating one for simplifying read-over-write expressions. In Sect. 10 we briefly mention related work. Finally we summarize in Sect. 11 and acknowledge the help of colleagues in Acknowledgements.

3 A Little Background on ACL2

In this section we present a little practical background on ACL2, its documentation and user-developed libraries. Then we sketch the syntax of the ACL2 logic and reveal a bit about the implementation of the ACL2 theorem prover in Lisp. We also reveal a bit about the semantics.

ACL2 was initially developed by Robert S. Boyer and the author starting in 1989. However, since the early 1990s it has been extensively further developed, documented, maintained, and distributed by Matt Kaufmann and the author. It is available for free in source code form from the ACL2 home page [14].

When we refer to ":DOC *x*" we mean the documentation topic *x* in the online ACL2 documentation, which may be found by visiting the ACL2 home page, clicking on The User's Manuals, then clicking on ACL2+Books Manual and typing *x* into the "Jump to" box.

In ACL2 parlance, a "book" is a file of definitions and theorems that can be loaded (see :DOC `include-book`) into an ACL2 session to extend the current theory. The actions of the ACL2 rewriter (and other parts of the prover) are influenced by previously proved theorems. Books are often developed with some particular problem domain and proof strategy in mind and when included in a session configure the prover to implement that strategy.

In this chapter we refer to several books in the *ACL2 Community Book Repository*. The repository is developed and maintained by the ACL2 user community. The top of the directory structure may be viewed by visiting GitHub at https://github.com/acl2/acl2. A particular file may be found by clicking your way down the directory hierarchy. For example, to find `books/projects/stateman/stateman22.lisp` start on the GitHub page above and click on `books`, then `projects`, etc.

ACL2 is the name of a programming language, a first order logic, a theorem prover, and a program/proof development environment. The ACL2 programming language is an extension of the applicative subset of Common Lisp [19]. The logic includes an axiomatization of that language consistent with Common Lisp. The theorem prover and environment are implemented (largely) in the ACL2 programming language.

In ACL2, the term $\mathtt{R}(a, n, st)$ is written `(R a n st)`. ACL2 is case insensitive so this could also be written `(r a n st)` or `(R A N ST)`. In this chapter we write variable symbols in lowercase italics. We tend to use case, both capitalization and uppercase, merely for emphasis. If our use of case and italics is confusing just ignore them!

Internal to the ACL2 theorem prover, the term `(R` *a n st*`)` is represented by the Lisp list that prints as `(R A N ST)`, i.e., a list of length 4 whose `car` or first element is the Lisp symbol `R`, and whose `cdr` or remaining elements are given by the list `(A N ST)`. Consing the symbol `R` onto the list `(A N ST)` produces the list `(R A N ST)`. In Lisp we could create this list by evaluating `(cons 'R '(A N ST))`, or `(cons 'R (list 'A 'N 'ST))` or `(list 'R 'A 'N 'ST)`. These three examples illustrate the most common idioms used to create terms when programming the theorem prover.

This brings us to the single quote mark and Lisp evaluation. The Lisp convention is that a single quote mark followed by a Lisp expression α is read as though the user had typed `(QUOTE` α`)`. Thus, `'(R A N ST)` is read as `(QUOTE (R A N ST))`.

`QUOTE` is a "special symbol" in the semantics of Lisp. The result of evaluating `(QUOTE` α`)` is α. This discussion of internal representation and the special meaning of `QUOTE` and the single quote mark are relevant to our discussion of metafunctions in the next section.

But to foreshadow that discussion, it happens that if α is the Lisp representation of an ACL2 term then `'`α is the Lisp representation of another ACL2 term, that second term in fact denotes a constant in the ACL2 logic, and there is an ACL2 function, say $\mathcal{E}$, called an "evaluator," that when applied to that constant and an appropriate association list ("alist") will return the same thing as the value of α. For example, since `(R` *a n st*`)` is an ACL2 term, then so is `'(R A N ST)`, the latter term denotes a constant in the ACL2 logic, and

$$(\mathcal{E}\ \texttt{'(R A N ST) (list (cons 'A}\ a\texttt{) (cons 'N}\ n\texttt{) (cons 'ST}\ st\texttt{)))}$$
$$=$$
$$\texttt{(R}\ a\ n\ st\texttt{)}$$

is a theorem of ACL2.

In Lisp, certain constants, in particular symbols `T` and `NIL`, numbers, character objects, and strings, evaluate to themselves. Thus, when writing Lisp it is not necessary to quote these constants. But constants appearing in ACL2 terms, even `T`, `NIL`, and numbers, are always quoted. This is achieved without inconveniencing the user by *translating* user type-in into ACL2's internal form. Thus, the term we write as `(R 4520 8` *st*`)` is represented inside the theorem prover as `(R '4520 '8 ST)` which we could display as `(R (QUOTE 4520) (QUOTE 8) ST)`. The user could in fact input the term in any of these ways. All three expressions produce exactly the same internal form. And because ACL2 is Lisp, it happens that all three are not only ACL2 terms but Lisp expressions and they produce the same results when evaluated by Lisp.

Some other ACL2 function symbols used in this chapter are shown in Fig. 1. In Lisp, a test or predicate is said to be "false" if its value is `NIL` and is said to be "true" otherwise. The symbols `force` and `hide` of Fig. 1 are trivial identity functions used to communicate pragmatic information to the ACL2 prover. See :DOC `force` and `hide`.

ACL2 term	*name*	*conventional notation*
`(if` x y z`)`	if-then-else	x ? y : z
`(implies` p q`)`	logical implication	$p \to q$
`(and` p q`)`	logical conjunction	$p \wedge q$
`(or` p q`)`	logical disjunction	$p \vee q$
`(not` p`)`	logical negation	$\neg p$
`(equal` x y`)`	equality	$x = y$
`(integerp` x`)`	"is-integer"	$x \in \mathbb{Z}$
`(natp` x`)`	"is-natural"	$x \in \mathbb{N}$
`(<` x y`)`	less than	$x < y$
`(<=` x y`)`	less than or equal	$x \leq y$
`(+` x y`)`	addition	$x + y$
`(-` x y`)`	subtraction	$x - y$
`(*` x y`)`	multiplication	$x \times y$
`(ifix` x`)`	"coerce-to-integer"	if x is an integer, x; else 0
`(expt` x y`)`	exponentiation	x^y
`(mod` x y`)`	modulus	$x \bmod y$
`(ash` x y`)`	shift	$\lfloor x \times 2^y \rfloor$
`(logand` x y`)`	bitwise and	$x \& y$
`(logior` x y`)`	bitwise inclusive or	$x \vert y$
`(logxor` x y`)`	bitwise exclusive or	$x\hat{}y$
`(force` x`)`		x
`(hide` x`)`		x
`(R` a n st`)`	read n bytes from addr a	
`(!R` a n v st`)`	write n bytes of v to addr a	

Fig. 1 Some ACL2 function symbols

In the internal representation of ACL2 terms, all function symbols take a fixed number of arguments. "Functions" that allow varying numbers of arguments are handled as Lisp macros that expand during the previously mentioned translation phase. For example, the internal form of `(+` i j k`)` is actually `(binary-+ i (binary-+ j k))`. The symbol + is a macro that expands into a term that uses the function symbol `binary-+`. Of the "function symbols" shown in Fig. 1 the symbols `+`, `*`, `logand`, `logior`, and `logxor` are actually macros that expand into right-associated calls of function symbols that take exactly two arguments. The "functions symbols" `and` and `or` are macros that expand into nests of `IF` expressions. But in this chapter we ignore such details and will pretend that they are all function symbols, not macros; when discussing term processing functions we will act like these symbols have exactly two arguments. We mention this detail only to reassure readers familiar with ACL2 that our metafunctions do not mistake macros for function symbols.

ACL2 is untyped and all ACL2 functions are total; thus, ACL2 expressions mean *something* no matter what well-formed arguments are supplied; however we will always use them conventionally and their completions are unimportant here. For example, ACL2's universe includes the rationals but not the irrationals. Thus, `(expt 2 1/2)` is a well-formed ACL2 term, it is indeed equivalent to a certain constant, but that constant is not $\sqrt{2}$. But this does not matter here because no term involved in this work applies `expt` to a non-integer.

4 Metafunctions

ACL2 "metafunctions" are ordinary ACL2 functions that operate on the internal representation of ACL2 terms. Correctness is stated in terms of "evaluators." Once ACL2 has proved a metafunction correct, the metafunction may be used by the theorem prover directly on the internal representation of terms [4]. Metafunctions have been part of ACL2 since its beginning; indeed, they were first introduced and described in 1979 [3] as part of the prover that became Nqthm [5].

An "evaluator" is a function that interprets an object as a term, with respect to some assignment giving meaning to variable symbols. Lisp's `eval` would be a wonderful evaluator if it were admissible in ACL2's first order logic of total recursive functions, but it is not. Fortunately, it suffices for ACL2's purposes to admit evaluators for a finite number of already-introduced function symbols and the ACL2 system provides a macro, `defevaluator`, that makes this easy. See :DOC `defevaluator`.

More technically, let σ be a set of ACL2 function symbols. An ACL2 *evaluator* function over σ is a function ev of two arguments, x, treated as the internal representation of a term, and $alist$, treated as an association list mapping variable symbols to values. The value, v, of `(`$ev\ x\ alist$`)` is constrained to have certain properties including: If x is a symbol other than `NIL`, v is the value assigned x by $alist$. If x is `'c`, v is c. If x is of the form $(g\ x_1\ \ldots\ x_n)$, where $g \in \sigma$, then v is `(`g `(`$ev\ x_1\ alist$`)` ... `(`$ev\ x_n\ alist$`))`. Additional constraints include that ev be able to interpret `LAMBDA`-applications and that on x of the form $(g\ x_1\ \ldots\ x_n)$ where $g \notin \sigma$, ev is a function of the `(`$ev\ x_i\ alist$`)`.

Henceforth, we will assume that $\mathcal{E}$ is an ACL2 evaluator function over all of the functions mentioned in this chapter (except $\mathcal{E}$ itself!).[3]

Thus,

```
(ℰ '(!R '4000 '8 (LOGAND X Y) ST)
   (LIST (CONS 'X x)
         (CONS 'Y y)
         (CONS 'ST st)))
=
(!R '4000 '8 (LOGAND x y) st)
=
(!R 4000 8 (LOGAND x y) st).
```

[3]The actual name of this evaluator is `stateman-eval`, "stateman" being the name of the "State Management" book that motivated this work. We simply find `stateman-eval` inconveniently long for use in a paper.

A *metafunction* is an ordinary list processing function in ACL2 with the property that it takes the internal representation of a term and returns the internal representation of an equivalent term. To be precise, a metafunction must be proved to operate correctly on “pseudo terms.” Pseudo terms are term-like list structures that do not necessarily obey all the internal invariants on ACL2’s term representation. Before the output of a metafunction is *used* to replace its input, the output is checked to satisfy all the internal invariants, unless the user has also proved that the function preserves them [15].

The general form of the theorem establishing that fn is a verified metafunction is:

```
(implies (and (pseudo-termp x)
              (alistp alist))
         (equiv (ℰ x alist)
                (ℰ (fn x mfc state) alist)))
```

where $\mathcal{E}$ is any evaluator. The variable name mfc stands for *metafunction context* and $state$ is the state of the ACL2 system, which together give fn access to contextual and heuristic data.

If this theorem has been proved by ACL2, then the ACL2 rewriter is logically permitted to replace any term x by the result computed by calling fn on x provided the returned object represents a term. This argument is presented in detail in :DOC `meta`.

Furthermore, by convention, if the metafunction returns an answer of the form `'(IF` *test new x*`)` when applied to x, the rewriter uses *new* as the simplified version of x provided it can backchain and establish $test$. Thus, `fn` can check some hypotheses syntactically and leave others to be relieved by the rewriter. This design means that the user does not have to prove that the metafunction properly interprets the data found in mfc and $state$. It also means that the ACL2 implementors do not have to formalize that data but instead merely provide functions for accessing certain parts of it. However, when those functions are used properly in a metafunction and the metafunction accurately “exports” what was learned as a conjunct included in $test$, it is generally easy for ACL2 to backchain and prove $test$: it is generally proved by the trusted internal routines of ACL2 for interpreting the data in mfc and $state$.

Since ACL2’s implementation language is ACL2, programming metafunctions is just like programming theorem proving utilities, except that we generally use ACL2 to prove that our programs are correct. For example, suppose we wanted a utility for conservatively determining that an expression x always returns a natural number. Here is such a function.[4] It is not actually necessary for the user to define this particular function. ACL2 has much more sophisticated built-in ways to recognize expressions that return naturals. But this function is a good warm-up.

[4]As indicated above, a correct definition will use `BINARY-+` instead of `+`, `BINARY-*` instead of `*`, etc.

```
(defun syntactic-natp (x)
  (cond
   ((atom x) nil)
   ((eq (car x) 'QUOTE)
    (natp (nth 1 x)))
   ((member (car x) '(+ * LOGAND LOGIOR LOGXOR ASH MOD))
    (and (syntactic-natp (nth 1 x))
         (syntactic-natp (nth 2 x))))
   ((eq (car x) 'HIDE)
    (syntactic-natp (nth 1 x)))
   ((eq (car x) 'R) t)
   (t nil)))
```

Here we use `atom` to recognize variable symbols, `(car` *x*`)` to fetch the top-level function symbol (or the `QUOTE` mark) of the non-atomic term *x*, `(nth 1` *x*`)` to fetch the constant inside a `QUOTE`d expression, and `(nth` *i x*`)` to fetch the i^{th} argument of function application *x*.

ACL2 can prove that if `(syntactic-natp` *term*`)` is true, then `(natp (`$\mathcal{E}$ *term alist*`))`.

```
(implies (syntactic-natp term)        ; {syntactic-natp correct}
         (natp (ℰ term alist)))
```

We might then use `syntactic-natp` in the definition of some metafunction. For example, suppose we wished to write a metafunction that recognized terms of the form `(natp` *x*`)` and replaced them by `T` when *x* is a `syntactic-natp` expression. Here is that metafunction:

```
(defun meta-natp (x)
  (cond ((and (not (atom x))
              (eq (car x) 'NATP)
              (syntactic-natp (nth 1 x)))
         '(QUOTE T))
        (t x)))
```

ACL2 can prove:

```
(implies (pseudo-termp x)                  ; {meta-natp correct}
         (equal (ℰ x alist)
                (ℰ (meta-natp x) alist)))
```

Given this theorem, ACL2 would be justified in applying `meta-natp` to every expression it ever encountered and replacing the expression by the result. That would be needlessly inefficient since `meta-natp` only changes some `NATP` expressions. The user-interface to ACL2 requires the user to provide pragmatic information identifying likely targets expressions, in this case, calls of `NATP`.

5 Bounders

The key to resolving such questions as $(a + n \leq b)$ by syntactic analysis is to be able to compute a bounded interval containing all possible values of a term. In this chapter we assume all intervals are closed, bounded, and over the naturals (i.e., integer intervals with non-negative lower bound). We denote intervals over the naturals by $[lo, hi]$, where both lo and hi are natural numbers and $lo \leq hi$.

Imagine that x and y lie within certain bounded closed intervals over the naturals. Then it is easy to compute an interval containing their sum by appealing to the following theorem:

$$(x \in [lo_x, hi_x] \wedge y \in [lo_y, hi_y]) \rightarrow (x + y) \in [lo_x + lo_y, hi_x + hi_y]$$

It is easy to imagine a function that takes a term, like `(+ x y)` in ACL2, and computes an interval containing its value, provided it can recursively compute such intervals for x and y. The question is: given intervals containing the arguments of a function f, can we compute an interval containing the value of f on those arguments?

In ACL2, an n-ary function g is a *bounder* for an n-ary function f if, for closed bounded intervals $int_1, int_2, \ldots, int_n$ over the natural numbers, when $x_i \in int_i$, for all $1 \leq i \leq n$, then $g(int_1, \ldots, int_n)$ is an interval and $f(x_1, \ldots, x_n) \in g(int_1, \ldots, int_n)$.[5]

The file `books/tau/bounders/elementary-bounders.lisp`, in the ACL2 Community Books repository, developed by the author, defines and verifies bounders for `+`, `*`, `-`, `FLOOR`, `MOD`, `LOGAND`, `LOGNOT`, `LOGIOR`, `LOGXOR`, `EXPT`, `ASH` and a few other functions.

For example, here is a version of the bounder for `LOGAND` that is correct provided the two intervals int_x and int_y are closed bounded intervals over the naturals. This function is less general than that in the `elementary-bounders` Community Book, which deals with the various kinds of ACL2 intervals, including the cases where the bounds are negative integers. But the simple function below illustrates the basic ideas in all of our bounders.

```
(defun natp-tau-bounder-logand (int_x int_y)
  (let ((lo_x (tau-interval-lo int_x))
        (hi_x (tau-interval-hi int_x))
        (lo_y (tau-interval-lo int_y))
        (hi_y (tau-interval-hi int_y)))
    (cond
     ((worth-computingp lo_x hi_x lo_y hi_y)
      (make-natural-interval
       (find-minimal-logand lo_x hi_x lo_y hi_y)
       (find-maximal-logand lo_x hi_x lo_y hi_y)))
```

[5] ACL2 is actually a little more relaxed: it does not require that every argument of f be confined to an interval. ACL2 furthermore allows both open and closed intervals, possibly unbounded at either end, over not just the integers but also the rationals.

```
     (t
      (make-natural-interval 0 (min hi_x hi_y))))))
```

Here the functions `tau-interval-lo` and `tau-interval-hi` extract the lower and upper bounds of an interval, and `make-natural-interval` constructs a closed ACL2 interval over the naturals when given appropriate lower and upper bounds. We discuss the "Tau System" of ACL2 in the next section.

The naive analytic bound on `(logand` x y`)` is $[0, \min(hi_x, hi_y)]$: the minimum possible value is `0` because x and y may not have any bits in common. The maximum possible value is the smaller of the upper limits of x and y, since `logand` just turns some bits off. For example, if $x \in [1032, 1039]$ and $y \in [520, 527]$, then this naive approach tells us that `(logand` x y`)` $\in [0, 527]$.

But this naive approach can grossly overestimate the bounding interval. In fact, `(logand` x y`)` $\in [8, 15]$, for any x and y bounded as assumed above, as can be confirmed by simply trying every combination of x and y in the two intervals and `logand`ing them together. If the two input intervals are sufficiently small this empirical approach is practical and often produces much tighter results. The functions `worth-computingp`, `find-minimal-logand`, and `find-maximal-logand` implement this empirical approach to interval analysis. `Worth-computingp` deems it worth trying if the number of combinations is less than 2^{20}. ACL2 can do that many `logand` operations in about 0.004371 s on a MacBook Pro laptop with a 2.6 GHz Intel Core i7 processor.

6 Ainni: Abstract Interpreter over Natural Number Intervals

The "easy to imagine" function mentioned above, that takes a term and tries to compute an interval containing its value, is formalized in our function `Ainni`. `Ainni` is an abstract interpreter over natural number intervals. It uses the bounders in the elementary bounders book, and a few more, compute intervals.

To suggest how `Ainni` is defined we exhibit a simpler function `aii` below. For the full definition of `Ainni` see the ACL2 Community Book `books/projects/-stateman/stateman22.lisp`.

Suppose we have k function symbols, op_1, ..., op_k, of arities n_1, ..., n_k, and suppose we have a bounder function for each, `bounder-`op_1, ..., `bounder-`op_k, respectively. Suppose x is a term over the op_i. Then here is a sketch of `aii`, an abstract interpreter that attempts to compute an interval containing the value of x. If it fails to find an interval it returns `nil`. We show the definition below and then paraphrase each case shown.

```
(defun aii (x)
  (cond
   ((atom x) nil)
   ((eq (car x) 'QUOTE)
    (cond ((natp (nth 1 x))
           (make-nat-interval (nth 1 x) (nth 1 x)))
          (t nil)))
 ...
 ((eq (car x) 'op_i)
    (let ((int1 (aii (nth 1 x)))
          ...
          (intn_i (aii (nth n_i x))))
      (cond
       ((and int1 ... intn_i)
        (bounder-op_1 int1 ... intn_i)
        (t nil)))))
 ...
   ((eq (car x) 'R)
    (cond ((and (not (atom (nth 2 x)))
                (eq (car (nth 2 x)) 'QUOTE)
                (natp (nth 1 (nth 2 x))))
           (make-nat-interval
            0
            (- (expt 2 (* 8 (nth 1 (nth 2 x)))) 1)))
          (t nil)))
   (t nil)))
```

If x is a variable symbol, `aii` fails and returns `nil`. If x is a natural number constant, 'k, it returns the interval $[k, k]$. If x is an application of one of the known op_i, `aii` recursively computes an interval for the n_i arguments and, provided it succeeds on each, it calls the bounder for op_i to compute the interval for the call. If x is an application of `R`, `aii` asks whether the extent is a natural number constant, `'`k, and if so returns $[0, 2^{8k} - 1]$. Otherwise, `aii` fails and returns `nil`.

Of course, the definition could be made more efficient by "failing early," e.g., not trying to compute an interval for the second argument if it failed to find one for the first. Furthermore, some terms can be bounded even if some of their arguments cannot be, e.g., `(logand` x `31)` $\in [0, 31]$ regardless of x's value. But `aii` is offered only as a suggestive model of our more sophisticated `Ainni`.

A more basic question arises when looking at the definition of `aii`: What about intervals for variables? The function above just fails if it encounters a variable. `Ainni` on the other hand takes another argument, called *ctx*, which provides contextual information, gleaned from the hypotheses governing the occurrence of the term x. For our purposes here, think of *ctx* as a map from Boolean terms to truth values. For example, the assumption that (`(R a 8 st)` < 16) would be coded in *ctx* as a pair associating the term `(< (R a 8 st) 16)` with true.[6] `Ainni` uses its *ctx* argument to determine the arithmetic bounds on variable values. In our application, the only "variables" encountered are actually reads from memory, i.e., expressions

[6]What we are calling *ctx* here is actually ACL2's "type-alist," and it pairs arbitrary terms with "types" gleaned from the governing hypotheses.

of the form `(R` *a n st*`)`. If the extent of the read is a natural number constant then `(R` *a* `'`*k st*`)` $\in [0, 2^{8k}-1]$. However, `Ainni` uses the *ctx* argument to try to narrow that interval by looking for assumptions on the bounds of `(R` *a* `'`*k st*`)`.

`Ainni` takes three inputs: the term *x* to bound, a list of hypotheses, *hyps*, assumed so far, and *ctx*. It returns three values. These values are formally written as shown below and have the following interpretations:

- `(mv-nth 0 (Ainni` *x hyps ctx*`))`: the 0th returned value of `(Ainni` *x hyps ctx*`)`. Informally this result is called the "output flag." When the output flag is non-`nil` ("true") it means `Ainni` successfully computed an interval for *x*; when the output flag is `nil`, `Ainni` could not find a suitable interval, e.g., perhaps the input term *x* is not in the set of terms recognized by `Ainni`. When the output flag is `nil`, the other two results are `nil` (and irrelevant).
- `(mv-nth 1 (Ainni` *x hyps ctx*`))`: the 1st returned value of `(Ainni` *x hyps ctx*`)`. Informally this result is called the "list of output hypotheses" and each element is called an "output hypothesis." When the output flag is non-`nil`, the list of output hypotheses is a list of terms that `Ainni` is relying on for the correctness of its answer. The output hypotheses include all of the elements of the input hypotheses *hyps* plus any hypotheses that `Ainni` extracted from *ctx* that contributed to its answer.
- `(mv-nth 2 (Ainni` *x hyps ctx*`))`: the 2nd returned value of `(Ainni` *x hyps ctx*`)`. Informally this result is called the "output interval." When the output flag is non-`nil`, the output interval is a bounded natural number interval and the value of *x* (under the evaluator $\mathcal{E}$ with any variable assignment *alist*) lies within the output interval, provided the value of each output hypothesis is true (under the same evaluator $\mathcal{E}$ with the same variable assignment *alist*).

Four important theorems about `Ainni` have been proved with ACL2. The first says that when given a pseudo term *x* and a list of pseudo terms *hyps* the output hypotheses are all pseudo terms.

```
(implies                                                ; {Ainni 1}
 (and (pseudo-termp x)
      (pseudo-term-listp hyps))
 (pseudo-term-listp
  (mv-nth 1 (Ainni x hyps ctx))))
```

The second theorem establishes that when `Ainni`'s output flag is non-`nil` its output interval is indeed a bounded interval over the naturals.

```
(implies                                                ; {Ainni 2}
  (and (pseudo-termp x)
       (mv-nth 0 (Ainni x hyps ctx)))
  (and (tau-intervalp
        (mv-nth 2 (Ainni x hyps ctx)))
       (equal (tau-interval-dom
               (mv-nth 2 (Ainni x hyps ctx)))
              'INTEGERP)
       (tau-interval-lo
        (mv-nth 2 (Ainni x hyps ctx)))
       (tau-interval-hi
        (mv-nth 2 (Ainni x hyps ctx)))
       (<= 0 (tau-interval-lo
              (mv-nth 2 (Ainni x hyps ctx)))))))
```

The first conjunct in the conclusion states that the output interval is an interval; the next conjunct states that the domain of the interval is `INTEGERP`. The next two conjuncts state that the lower and upper bounds of the output interval are non-`nil`, which (because of the first two conjuncts) means they are both integers, the lower bound is weakly below the upper one, and the interval is closed.[7]

The third theorem establishes that for pseudo term x such that `Ainni`'s output flag is non-`nil` and all of the output hypotheses are true (i.e., the evaluator $\mathcal{E}$ evaluates the conjunction of those terms to non-`nil`), then the value (under $\mathcal{E}$) of x is contained in the output interval.

```
(implies                                                    ; {Ainni 3}
 (and (pseudo-termp x)
      (mv-nth 0 (Ainni x hyps ctx))
      (ℰ (conjoin (mv-nth 1 (Ainni x hyps ctx)))
        alist))
 (in-tau-intervalp (ℰ x alist)
                   (mv-nth 2 (Ainni x hyps ctx))))
```

The fourth theorem establishes that `Ainni` actually preserves the internal invariants on ACL2 terms, i.e., that if the input term and the elements in the input hypotheses each satisfy ACL2's internal invariant then the output hypotheses satisfy that invariant. The constant `*stateman-arities*` is an alist pairing each of the function symbols known to $\mathcal{E}$ with its arity.

```
(implies                                                    ; {Ainni 4}
 (and (termp x w)
      (term-listp hyps w)
      (arities-okp *stateman-arities* w))
 (term-listp
  (mv-nth 1 (Ainni x hyps ctx))
  w))
```

This last theorem allows ACL2 to avoid checking that the output hypotheses satisfy the internal invariants on terms. Instead, ACL2 just has to check that each of the function symbols listed in `*stateman-alist*` has the given arity in ACL2's then-current logical theory.

`Ainni` is closely related to the Tau System in ACL2. See :DOC `tau-system`. Tau is a user extensible abstract interpreter over sets of monadic predicates describing the types of values returned by an expression. It includes containment in constant

[7]By definition of `tau-intervalp`, any interval with `INTEGERP` domain has integers for its bounds unless there is no bound (i.e., a "bound" of `nil`) in some direction. Furthermore, all bounded integer intervals are, by convention, closed. That is, if the domain is `INTEGERP` then instead of, say, [0,8) we use [0,7].

intervals as a “type.” ACL2 users think of the Tau System as a quick, incomplete “type checker” for the untyped language of ACL2. By design, the Tau System answers yes/no questions: is this formula trivial by type-like reasoning?

`Ainni` is designed to answer quantitative questions: What are the minimal and maximal values of this expression? `Ainni` exploits some of the same theorems (in the elementary bounders book) used to extend Tau. But by defining `Ainni` in the logic and verifying it, we make it possible to use interval reasoning during rewriting.

7 Some Examples

Consider this expression:

```
(+ 2000 (* 8 (LOGAND 31 (R 1000 8 st)))).
```

This is the formal expression of a fairly typical machine address encountered in symbolic code evaluation. It corresponds to the compiled version of an array element reference, where the base address of the array is `2000`, the array consists of quadword (8-byte) elements, and the index is formed by taking the bottom 5 bits of the quadword at address `1000`. The prevalence of constants in the expression is also quite common when exploring code recovered from an actual machine image: the locations of data are fixed or at computed offsets from fixed addresses like the initial stack pointer.

What can `Ainni` tell us about this expression? We answer that by evaluating

```
(Ainni '(+ 2000 (* 8 (LOGAND 31 (R 1000 8 st)))) nil nil)
```

Ainni will return three values. Its output flag will be `T`, the list of output hypotheses will be `nil`, and the output interval will be the ACL2 data structure that represents the integer interval [2000, 2248].[8]

The derivation of the output interval is as follows: `(R 1000 8 st)` is known to be in $[0, 2^{64} - 1]$, but the `LOGAND` is in [0, 31]. Thus, the product with `8` is in the interval [0, 248], so the sum with 2000 is in [2000, 2248].

Now imagine *ctx* contains the assumption that `(R 1000 8 st)` is below 16 and reconsider

```
(Ainni '(+ 2000 (* 8 (LOGAND 31 (R 1000 8 st)))) nil ctx).
```

This time the output flag will be `T`, there will be one output hypothesis, namely `(<= (R 1000 8 st) 15)`, and the output interval will be [2000, 2120]. The third correctness theorem for `Ainni` assures us that `(+ 2000 (* 8 (LOGAND 31 (R 1000 8 st))))` lies in [2000, 2120] provided `(<= (R 1000 8 st) 15)` is true.

[8] As noted earlier, the actual input to `Ainni` should be in ACL2’s internal form, so, for example, the “+” should be `binary-+` and the numbers should be quoted. The data structure representing the output interval is `(INTEGERP (NIL . 2000) . (NIL . 2248))`, indicating an integer domain, bounded above and below by `2000` and `2248` respectively. The `NIL`s indicate that $\leq$ rather than $<$ is used to check whether a number is in bounds.

Finally, to demonstrate Ainni's speed compared to ACL2's more powerful arithmetic, consider the expression

```
(LOGIOR (ASH (MOD (R 1000 4 ST) 2) 0)
        (ASH (MOD (R 1004 4 ST) 2) 1)
        (ASH (MOD (R 1008 4 ST) 2) 2)
        ...
        (ASH (MOD (R 1052 4 ST) 2) 13)
        (ASH (MOD (R 1056 4 ST) 2) 14)
        (ASH (MOD (R 1060 4 ST) 2) 15)).
```

The value of this expression lies in the interval $[0, 2^{16} - 1]$ regardless of the values of the R-expressions. Any programmer would realize the expression is bounded above by 2^{16}: each MOD is just a single bit, and the expression shifts those bits into positions 0–15. Using similar "forward" reasoning from the expression, Ainni computes the interval $[0, 2^{16} - 1]$ in 0.012 s on a MacBook Pro laptop with a 2.6 GHz Intel Core i7 processor running ACL2 in CCL.

On the other hand, *proving* the expression is so bounded can feel harder! Indeed, it takes the same laptop about 1306 s to use the standard ACL2 arithmetic library from the Community Books (books/arithmetic-5/top) to prove that the expression above is less than 2^{16}. The library splits the goal into 2^{16} cases.

Of course, ACL2's arithmetic library is much more powerful than Ainni. The library is essentially a collection of theorems about arithmetic/logical functions which informs the ACL2 rewriter and its integrated linear arithmetic decision procedure. Those systems can be made to prove anything that is provable about ACL2 arithmetic, whereas Ainni is much more limited. But we embarked on the development of Ainni because we saw the importance of a verified tool to look at typical machine arithmetic expressions and do what every programmer can do: bound it by interval reasoning. In addition, Ainni is fast.

The expression above is small compared to expressions encountered when doing code analysis, especially of long sequences of machine instructions. The expression above has 63 function calls in it (when the LOGIOR macro is expanded into a right-associated nest of calls of BINARY-LOGIOR) of which 16 are calls to R and the rest are calls to logical functions. By contrast, the largest arithmetic/logical expression encountered in the disassembly of a machine code implementation of the DES algorithm is a term involving 147,233 function applications, 31,361 of which are calls of R and the rest are calls of arithmetic/logical functions. Ainni can bound that very large expression in about 0.01 s. It is completely impractical to use the standard arithmetic library to confirm the correctness of that answer (other than by relying on the verified correctness of Ainni).

But another major advantage of Ainni, aside from being very fast and quite capable on huge expressions, is that it *discovers* bounds whereas the rest of ACL2 (e.g., the Tau System) is oriented toward *proving* things. That is, ACL2 is generally used to answer specific Boolean questions, e.g., "Does this value fit in 16-bits" whereas Ainni gives it the capability of answering quantitative questions such as "How big is this?" These advantages mean `Ainni` can effectively be used in simplification.

8 Using Ainni in a Metafunction

Because `Ainni` has been proved correct by ACL2, it can be used in metafunctions which are in turn used by the rewriter. Thus, one need not choose between `Ainni` and a conventional rewrite-driven arithmetic book; one can have both.

Here is a very simple metafunction that shows how we use `Ainni`. The following function simplifies `(MOD` *x* `'`*k*`)` expressions, where *k* is some natural constant, using the fact that `(MOD` *x* `'`*k*`)` $= x$, if *x* is an integer less than *k*.

```
(defun mod-constant-simplifier (term mfc state)
  (declare (ignore state))
  (cond
   ((and (not (atom term))
         (eq (car term) 'MOD)
         (not (atom (nth 2 term)))
         (eq (car (nth 2 term)) 'QUOTE))
    (let ((x (nth 1 term))
          (k (nth 1 (nth 2 term)))
          (ctx (mfc-type-alist mfc)))
      (cond
       ((and (natp k)
             (syntactic-natp x))
        (mv-let
         (flg hyps int)
         (Ainni x nil ctx)
         (cond
          ((and flg
                (< (tau-interval-hi int) k))
           (list 'IF (conjoin hyps) x term))
          (t term))))
       (t term))))
   (t term)))
```

This function checks that *term* is a call of `MOD` and that the second argument is a quoted constant. If so, it binds *x* to the first argument of the `MOD` and *k* to the constant. It also extracts the type-alist from the metafunction context *mfc* and binds the variable *ctx* to that. Then it checks that *k* is a natural number and *x* is a syntactic natural. If so, it calls `Ainni` and if `Ainni` reports success and the upper bound of the resulting interval is below *k*, it creates and returns an `IF`. The test of the `IF` is the conjunction of the output hypotheses, the true branch is *x*, and the false branch is *term*.

The correctness of this metafunction follows from the correctness of `syntactic-natp` and `Ainni` and the previously mentioned fact about `(MOD` *x* `'`*k*`)`. Once verified and installed as a metafunction for `MOD`, `mod-constant-simplifier` is run on every `MOD` expression and, when it returns something different from its input, the theorem prover backchains to establish the truth of the tested output hypotheses and if so replaces the target term with *x*.

For example, once `mod-constant-simplifier` is verified as a metafunction for `MOD`, the expression:

```
(MOD (LOGIOR (ASH (MOD (R 1000 4 ST) 2) 0)
             (ASH (MOD (R 1004 4 ST) 2) 1)
             (ASH (MOD (R 1008 4 ST) 2) 2)
             ...
             (ASH (MOD (R 1052 4 ST) 2) 13)
             (ASH (MOD (R 1056 4 ST) 2) 14)
             (ASH (MOD (R 1060 4 ST) 2) 15))
     (EXPT 2 24))
```

immediately simplifies to the `LOGIOR` expression.

9 Other Uses of Ainni

While `Ainni` was developed for answering questions about machine addresses it is generally useful for answering quantitative questions about formal machine arithmetic as illustrated in the previous section.

Another very helpful use of `Ainni` is in a metafunction to simplify `(<` *x y*`)`. Triggered by the less than operator, <, the metafunction uses `Ainni` on *x* and *y* and if `Ainni` succeeds the metafunction can use quick checks on the endpoints to often reduce the `(<` *x y*`)` to `T` or to `NIL`. The comparable reduction by the native rewriter and its linear arithmetic procedure involves duplication of effort, essentially trying to rewrite both the inequality and its negation since only Boolean questions can be asked of them.

The motivating applications for `Ainni` were metafunctions to handle read-over-write and write-over-write expressions. Consider a read-over-write. Typically, the write expression is a deep nest of `!R` expressions. The metafunction uses `Ainni` on the read address and extent to compute the interval containing the region to be read. Then with that interval in hand it searches down the nest of writes comparing the read interval to the write intervals (using `Ainni` on each write address and extent). Quick checks on the resulting intervals can determine when the regions are disjoint – without having to reanalyze the addresses to determine whether the read is "above" or "below" the write.

Thus, `Ainni` allows the read-over-write metafunction to be much more efficient than rewrite rules because the read address and each write address is analyzed just once. This illustrates a major advantage of being able to answer a quantitative question rather than just a Boolean one.

ACL2 permits memoization and that has proven helpful in avoiding repeated calls to `Ainni` on the write addresses. However, we found that it was best to memoize the read-over-write metafunction itself rather than the individual calls of `Ainni` inside it.[9]

The details of the metafunctions using `Ainni` may be found by looking at the heavily commented proof script in the ACL2 Community Book `books/-projects/stateman/stateman22.lisp`.

10 Related Work

Simplification and abstract interpretation are so ubiquitous it is beyond the scope of this chapter to offer much background on them. Basically every mechanized prover has libraries or tactics or built-in routines to simplify formulas using various standard heuristics to control inference; see "auto" in Coq and HOL and the built-in notion of "simplification" in PVS. The name "abstract interpretation" was introduced by the Cousots in 1977 [8] and is basically the generalization of an operational semantics or interpreter to deal with conservative approximations of the actual data (e.g., intervals instead of numbers). Type checking is an example of abstract interpretation.

The work most closely resembling that reported here is probably the Astrée static analyzer [9]. Astrée aims at proving the absence of run time errors in programs written in C. It is based on abstract interpretation and uses interval analysis to approximate numeric data values.

However, Astrée is a standalone static analyzer whose input is a C program, whereas `Ainni` is a user-defined and mechanically verified extension of the ACL2 simplifier. While both are relying on abstract interpretation, Astrée interprets C programs (including its arithmetic/logical expression language) while `Ainni` only interprets arithmetic/logical expressions in the ACL2 logic. The program control and data manipulation done by Astrée is, in our case, done by the ACL2 system, specifically its simplifier applied to the formal operational semantics and the object code. So there are really two different abstract interpreters involved in our code proofs, one over the program text (done by the simplifier) and one over the semantics of arithmetic/logical expressions (done by `Ainni`), and in Astrée they are combined. One presumes that Astrée contains an abstract interpreter for arithmetic/logical expressions that produces interval bounds on those expressions.

[9]One could memoize the ACL2 rewriter itself and hope to speed up the rewrite-rule approach. However this has been unsuccessful because the ACL2 rewriter takes so many arguments to record the context, the objective of the rewrite, equivalence relations to be maintained, histories used to avoid infinite backchaining and looping, stacks to track the lemmas used for reporting purposes, counters to measure or limit the work done, etc. All these extra arguments mean that identical calls to rewrite virtually never occur and so memoization costs more time than it saves. `Ainni` and its callers use far fewer arguments and memoization is effective on them.

Finally, `Ainni` is available as a mechanically verified extension of the ACL2 simplifier and is hence of use in any theorem proving setting requiring reasoning about the bounds of arithmetic/logical expressions. Furthermore, `Ainni` has been mechanically verified to be correct by ACL2.

11 Conclusion

We have described an ACL2 function, `Ainni`, for answering the quantitative question "what are the minimal and maximal magnitude of the value of this expression?" The function is an abstract interpreter for machine arithmetic expressions composed of arithmetic/logical operators and interprets them over bounded closed natural number intervals. `Ainni` can be thought of as a type inference procedure where the types are intervals.

`Ainni` has been verified with ACL2 to be correct and can therefore participate in formal proofs. The vehicles for that participation are metafunctions designed to simplify machine arithmetic expressions.

`Ainni` has allowed ACL2 to do symbolic exploration of sequences of realistic machine code containing thousands of instructions, whose end states contain millions of function applications. This was not possible using other techniques we have tried with ACL2.

The success of `Ainni` has raised important new questions: how can the rest of ACL2 be made to cope with the expressions now being produced? This is a welcome — and very typical — step along the evolutionary path ACL2 has followed. A solution to one scaling problem introduces new scaling challenges.

Acknowledgements I would especially like to thank Warren Hunt for his invaluable help during the development of `Ainni`. Warren developed the definitions and proved many of the basic rewrite rules for the byte addressed read and write functions, `R`, and `!R`. He also provided an ACL2 formalization of a realistic ISA and implemented the DES algorithm in ACL2. We then compiled the DES algorithm into the instructions of the ISA thus obtaining an interesting symbolic evaluation challenge for ACL2. I would also like to thank Matt Kaufmann, who gave me some strategic advice on lemma development to prove the correctness of one of the metafunctions here as well as his usual extraordinary efforts to maintain ACL2 while I pursue topics such as this one. This work was partially supported by ForrestHunt, Inc.

References

1. Bevier, W.R., Hunt Jr., W.A., Moore, J.S., Young, W.D.: Special issue on system verification. J. Autom. Reason. **5**(4), 409–530 (1989)
2. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, New York (1979)
3. Boyer, R.S., Moore, J.S.: Metafunctions: Proving them correct and using them efficiently as new proof procedures. Technical Report CSL-108, SRI International (1979)

4. Boyer, R.S., Moore, J.S.: Metafunctions: Proving them correct and using them efficiently as new proof procedures. The Correctness Problem in Computer Science. Academic Press, London (1981)
5. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook, 2nd edn. Academic Press, New York (1997)
6. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. J. ACM **43**(1), 166–192 (1996)
7. Brock, B., Kaufmann, M., Moore, J.S.: ACL2 theorems about commercial microprocessors. In: Srivas, M., Camilleri, A. (eds.) Formal Methods in Computer-Aided Design (FMCAD'96). LNCS, vol. 1166, pp. 275–293. Springer, Heidelberg (1996). https://www.cs.utexas.edu/users/moore/publications/bkm96.pdf
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min, A., Monniaux, D., Rival, X.: The astre analyser. In: Sagiv, M. (ed.) European Symposium on Programming (ESOP 2005). LNCS, vol. 3444, pp. 21–30. Springer, New York (2005)
10. Goel, S., Hunt, W.A., Kaufmann, M.: Simulation and formal verification of x86 machine-code programs that make system calls. In: Claessen, K., Kuncak, V. (eds.) FMCAD'14: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design, pp. 91–98. EPFL, Switzerland (2014)
11. Kaufmann, M., Manolios, P., Moore, J.S. (eds.): Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Press, Boston (2000)
12. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston (2000)
13. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common lisp. IEEE Trans. Softw. Eng. **23**(4), 203–213 (1997)
14. Kaufmann, M., Moore, J.S.: The ACL2 home page. Department of Computer Sciences, University of Texas at Austin (2014). http://www.cs.utexas.edu/users/moore/acl2/
15. Kaufmann, M., Moore, J.S.: Well-formedness guarantees for ACL2 metafunctions and clause processors. In: Design and Implementation of Formal Tools and Systems (DIFTS) (2015)
16. Liu, H., Moore, J.S.: Java program verification via a JVM deep embedding in ACL2. In: Slind, K., Bunker, A., Gopalakrishnan, G. (eds.) 17th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2004. Lecture Notes in Computer Science, vol. 3223, pp. 184–200. Springer, New York (2004)
17. Moore, J.S., Martinez, M.: A mechanically checked proof of the correctness of the Boyer-Moore fast string searching algorithm. In: Engineering Methods and Tools for Software Safety and Security (Proceedings of the Martoberdorf Summer School, 2008), pp. 267–284. IOS Press (2009)
18. Slobodova, A., Davis, J., Swords, S., Hunt Jr., W.: A flexible formal verification framework for industrial scale validation. In: Singh, S. (ed.) 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 89–97. IEEE (2011)
19. Steele Jr., G.L.: Common Lisp The Language, 2nd edn, p. 01803. Digital Press, Burlington (1990)
20. Toibazarov, E.: An ACL2 proof of the correctness of the preprocessing for a variant of the Boyer-Moore fast string searching algorithm. Honors thesis, Computer Science Dept., University of Texas at Austin (2013). See www.cs.utexas.edu/users/moore/publications/toibazarov-thesis.pdf
21. Wilding, M.: A mechanically verified application for a mechanically verified environment. In: Courcoubetis, C. (ed.) Computer-Aided Verification – CAV '93. Lecture Notes in Computer Science, vol. 697. Springer, Heidelberg (1993)

Engineering a Formal, Executable x86 ISA Simulator for Software Verification

Shilpi Goel, Warren A. Hunt Jr. and Matt Kaufmann

Abstract Construction of a formal model of a computing system is a necessary practice in formal verification. The results of formal analysis can only be valued to the same degree as the model itself. Model development is error-prone, not only due to the complexity of the system being modeled, but also because it involves addressing disparate requirements. For example, a formal model should be defined using simple constructs to enable efficient reasoning but it should also be optimized to offer fast concrete simulations. Models of large computing systems are themselves large software systems and must be subject to rigorous validation. We describe our formal, executable model of the x86 instruction-set architecture; we use our model to reason about x86 machine-code programs. Validation of our x86 ISA model is done by co-simulating it regularly against a physical x86 machine. We present design decisions made during model development to optimize both validation and verification, i.e., efficiency of both simulation and reasoning. Our engineering process provides insight into the development of a software verification and model animation framework from the points of view of accuracy, efficiency, scalability, maintainability, and usability.

1 Introduction

The development of large computing systems is almost always preceded by a modeling and analysis stage. When building a complex computing system, success is often determined by the accuracy of its model and the quality of analysis this model receives. For example, it is now impossible to design and manufacture a micro-

S. Goel (✉) · W.A. Hunt Jr. · M. Kaufmann
Department of Computer Science, The University of Texas at Austin,
2317 Speedway, M/S D9500, Austin, TX, USA
e-mail: shigoel@cs.utexas.edu

W.A. Hunt Jr.
e-mail: hunt@cs.utexas.edu

M. Kaufmann
e-mail: kaufmann@cs.utexas.edu

processor chip without extensive modeling, simulation, and product validation. The rising complexity and use of software systems necessitates building scalable tools for analyzing program behavior. To this end, we have used a rigorous mathematical approach to specify the x86 instruction-set architecture (ISA) for the formal analysis of x86 machine-code programs. Our specification of the x86 ISA is regularly validated to increase trust in the applicability of the results of formal analysis.

In principle, our approach to specifying an instruction-set architecture was known by von Neumann [1], Turing [2], and Zuse [3]. Our x86 ISA specification is simply a transition function that transforms one ISA-level state into the next one; this operation can be repeated to simulate any x86 program. A critical factor that makes the x86 ISA difficult to specify is its sheer size and complexity. The Intel manuals [4] that describe the x86 ISA are large documents consisting of around 3700 p. The x86 architecture continues to evolve. In fact, since we began our project in 2010, the x86 ISA has been significantly extended [5] — the AVX (Advanced Vector Extensions) is one such extension. This continued evolution is common for industrial products, which are modified and extended as business requirements demand.

In light of the size and complexity of the x86 ISA, the system chosen for its specification plays an important role. This system should be capable of rigorously specifying an ISA as large as the x86, and this specification should scale well and be maintainable. This system should offer an environment for both formal reasoning and validation; the system should provide a mathematical logic to support formal analysis about both the model itself and the x86 programs simulated on it, and the system's logic should be executable so that the ISA specification can be validated by performing *co-simulations* (i.e., comparisons of simulations of x86 programs on the model with corresponding executions on a real x86 processor). We use the ACL2 theorem-proving system [6, 7] to model the x86 ISA. ACL2 is a state-of-the-art system used in industry, and we have previously used ACL2 and its predecessor NQTHM [8] to model microprocessors at the RTL and ISA levels [9, 10].

Our goal is to ensure software reliability, and constructing a formal model of the x86 ISA is simply a means to that end. However, engineering a model of this scale is hardly a trivial undertaking. The model-building process is inherently error-prone because of the size and complexity of the ISA, and it is complicated further by the nature of demands placed on it. For example, to facilitate faster co-simulations, the model must provide high execution efficiency, and to support scalable formal analysis, the model must be defined using simple constructs that are easy to reason about. There is a tension between these two demands — optimizing a function for execution efficiency often increases the complexity of its underlying algorithm, making it more difficult to reason about the function. Another example of opposing demands, typical of large software projects, is completeness versus maintainability. Extending the model by supporting more x86 features and instructions should not make it unwieldy and difficult to maintain. Every design decision taken while building the model has to factor in such inherently disparate requirements. In this chapter, instead of focusing on our work on program verification, we discuss the framework that we built to attain our goal. We present our formal, executable model of the x86 ISA from an engineering standpoint and describe our design decisions, big and small, in detail.

Our x86 ISA model is more than 45,000 lines of executable ACL2 logic, with around 20,000 lines more for our libraries that aid in co-simulations and formal analysis. Our current focus is on the 64-bit mode[1] of the x86 ISA; in the rest of this chapter, whenever we refer to our x86 model, we mean our ISA model of the x86 processor in this 64-bit mode. The simulation speed of our model is either around 330,000 instructions/second or 3.3 million instructions/second, depending on its mode of operation.[2] To our knowledge, this is the fastest formal simulator for this complex an ISA model. We currently have a specification of 407 machine opcodes that include supervisor-mode and floating-point instructions. We have specified access control (via privileges and permissions) and address translation by capturing IA-32e paging and segmentation. We model the single-processor x86 architecture, but we someday hope to have a multi-core version. Just as the x86 ISA has been evolving since its introduction by Intel, our model continues to evolve as we support more features and instructions. Our x86 ISA model and related libraries for formal analysis and simulation are freely available [12] with a permissive BSD license. Their documentation [13] is available online too.

This presentation of our evolving x86 model is meant to be self-contained. We provide relevant details about ACL2 and the x86 ISA when needed. The reader can start with only the notion that a state-machine transformer will be presented, one which can be used both for simulation and for reasoning.

We begin in Sect. 2 by describing our modeling approach. The next two Sects. 3 and 4 present in some detail our model and its validation. After briefly describing in Sect. 5 the use of our model for software verification, we close in Sects. 6 and 7 with related work and concluding remarks.

2 Approach

We take a three-pronged approach to the construction of a model to be used for software verification. In Sect. 2.1, we explain our decision to use an interactive theorem prover as the choice of formal tool. Section 2.2 discusses why we perform analysis at the level of machine code to ensure software reliability. Having discussed the tool and the target of analysis, Sect. 2.3 sets out design goals for our framework, which includes a unified model for both simulation and formal analysis.

2.1 Formal Methodology

Our aim is to develop capabilities for thorough analysis of software to ascertain if, under a given set of conditions, it behaves as expected during run-time. Even

[1]Intel's IA-32e mode [11] of operation offers two sub-modes: the compatibility mode (which is similar to 32-bit protected mode) and 64-bit mode.

[2]The simulation speed is discussed in Sect. 4.

trivial programs rely on many complex conditions and properties for their correct execution. Such properties demand formal scrutiny, simply due to their complexity. Some examples are:

- The stack or the heap should not overwrite the program in any execution.
- The operating system kernel data should be invisible to application programs.
- Segmentation and paging data-structures should be set up correctly by the OS kernel. This is a property that is critical to establishing many others; e.g., a correct configuration will help in determining if process isolation is assured.

The thorough analysis of software requires capturing the *intent*, including the preconditions and expected behavior of the program. Fully automatic program analysis tools, like static and dynamic analyzers, do not impose this requirement of writing specifications on the user, but as a result, they are narrow in their scope. Though successful in carrying out their particular functions, like detecting memory safety violations, these tools do not support even the statement of many complex properties. In addition to such properties, useful metrics like program memory and space usage are difficult, if not impossible, to obtain using these tools.

The development of the CLI stack [14] in the 1980s using NQTHM, a Boyer–Moore theorem prover, has already demonstrated that theorem proving techniques alone can be applied to obtain various kinds of guarantees about computing systems. We will utilize the state-of-the-art in formal verification by coupling interactive theorem proving with automated tools, for example SAT solvers, to facilitate the development and delivery of secure systems.

2.1.1 ACL2

Our theorem prover of choice is the ACL2 system [6, 7], a descendant of the Boyer–Moore theorem prover used in the CLI stack project. It is a mathematical logic (a first-order logic of recursive functions) based on an applicative subset of Common Lisp as well as a mechanical theorem prover used to prove theorems in that logic. ACL2 is also a programming language, which offers the execution efficiency provided by underlying Lisp compilers, thereby enabling the construction of efficient executable formal models.

The ACL2 theorem prover has many automated proof strategies, and the application of a proof strategy recursively decomposes goals until all subgoals are proved. Users can extend the prover by adding their own proof strategies and external deduction tools, like SAT/SMT solvers, by using *clause processors* [15]. We often use GL [16], a framework for proving theorems involving finite objects using either a formally verified BDD utility or an external SAT solver like Glucose [17, 18].

ACL2 and its libraries, called *books*, are freely available [19] and include extensive documentation, with over 10,000 topics [20, 21]. These books, developed over many years, are typically included in ACL2 projects to build on existing definitions and theorems. Books can be certified by ACL2 to ensure their soundness.

ACL2's authors are the recipients of 2005 ACM Software System Award [22]. ACL2 is an industrial-strength system; it is regularly used in both academic and commercial applications [23]. For example, Centaur Technology [24], an x86-compatible processor vendor, uses ACL2 every day for modeling and verifying Centaur processor designs [25–27]. Other organizations with significant usage of ACL2 for formal verification are AMD [28–31], IBM [32–35], Intel [36], Kestrel [37, 38], Oracle [39], and Rockwell-Collins [40–42].

We now briefly discuss some ACL2/Lisp features that are especially important when building large formal and executable models. Some efficient data structures provided by ACL2 are discussed in Sects. 3.1 and 3.2.

Guards Though Lisp is syntactically untyped, the Common Lisp standard [43] specifies an intended domain for each Lisp primitive. The behavior of those primitives outside these domains depends on the particular Lisp implementation.

In ACL2, a mechanism called *guards* [44] is used to specify the intended domain of a function. *Completion axioms* define the behavior of a primitive for every possible input; thus, all ACL2 functions are logically total. When the guards of a function `f` are verified, it means that `f` respects the guards of all functions used in its body. A guard-verified function can thus be called natively in the host Lisp (as opposed to evaluation within ACL2), such that every ensuing call of any function `g` respects the guard of `g`. Thus, verification of the guards of a function improves its execution speed. Guards have no effect during reasoning — they do not affect the logical axiom that is added when an ACL2 function is defined.

Another feature that can allow faster execution of ACL2 functions is `mbe` [45] ("must be equal"). This construct allows the following form to be used in functions:

```
(mbe :logic <logic-code> :exec <exec-code>)
```

Users can choose to write simple code suitable for efficient reasoning in the `:logic` part and code optimized for execution efficiency in the `:exec` part. Logically, this `mbe` form is equal to `<logic-code>`. During execution, this form is equal to `<exec-code>` if the function's guards are verified. The `mbe` form generates proof obligations that are added to the guard proof obligations; when proved, they establish the equality of `<logic-code>` and `<exec-code>`, thereby allowing `<exec-code>` to be executed in place of `<logic-code>` when the function can be safely run natively in the host Lisp.

Type Declarations Common Lisp supports arbitrary-precision arithmetic; it imposes no limit on the magnitude of a number [46]. Machine integers, called *fixnums*, are efficiently represented and fast built-in arithmetic operations can be used when working with them. Integers with an unlimited number of bits are called *bignums* and special arithmetic functions are required when working with them. If an integer gets too big (where the "bigness" is defined by the Common Lisp implementation), Lisp promotes it from a fixnum to a bignum automatically and subsequent operations involving these numbers are the slower bignum operations. Therefore, avoiding bignum operations when possible is important for simulation efficiency.

Annotating ACL2/Lisp code with *type declarations* can avoid run-time type checking. Type declarations are a Common Lisp mechanism that allows the user to transmit information about types of objects to the underlying Lisp compiler. E.g., the user might declare that a certain variable, say `x`, is always of type `(unsigned-byte 32)`, i.e., an unsigned integer of width 32 bits. If the Lisp implementation's most positive fixnum is greater than or equal to $2^{32} - 1$ (true for 64-bit Lisp implementations), this type declaration enables the compiler to generate efficient machine code that does not involve either bignum computations or run-time type-checks.

In ACL2, type declarations need to be justified. They are added to the guard proof obligation to ensure that the execution of the function with type declarations in ACL2 can be safely done in the host Lisp without the possibility of encountering any run-time type violations.

Other Efficiency Concerns Two other efficiency-related issues need to be mentioned here — avoid consing (creation of ordered-pair data structures) and use inlining. Consing is a potentially expensive operation that allocates memory on the heap; avoiding it can substantially increase ACL2/Lisp code's execution efficiency.

Defining many small functions, rather than a single large function, is preferred for reasoning because it decomposes functionality into small easy-to-understand abstractions. However, for execution, function call overhead is high. Inlining small functions can improve the code's execution performance, but inlining big functions can cause code bloat. The decision to inline a function must be made judiciously.

Constrained Functions ACL2 provides a mechanism called `encapsulate` to introduce *constrained or uninterpreted functions*. The only things known about a constrained function are its name, signature (arity), and constraints, if any. Constrained functions can be introduced together with logical properties of those functions; but unlike defined functions, they are non-executable.

Trust Tags ACL2 allows users to extend the prover in ways otherwise not allowed by introducing a *trust tag* [47]. Typically, users extend the prover by creating ACL2 books that perform a potentially dangerous evaluation. An example of a dangerous evaluation is a function `foo` that invokes the command `date` on the underlying OS and returns the output; this is not a function in the logical sense because it may return different outputs for the same input (and in sound logics, `foo() == foo()` is always true). Books containing functions like `foo` must be certified with a trust tag. Note that careful use of `foo` (e.g., in a way that does not interfere with reasoning) should still be safe. If a book uses a trust tag, then it means that the functionality provided by the book is *trusted*; it is acknowledged that the formulas proved using that book are valid only if the book's functionality is correct. Among its other uses, trust tags allow the integration of various external tools [15] with ACL2.

Untouchable Functions ACL2 functions can be made *untouchable* [48] — it is syntactically illegal to call an untouchable function directly. Typically, this is done to preserve abstractions; e.g., if a top-level function calls several inferior functions, but the top-level function needs to be perceived as an atomic operation, then the inferior functions are made untouchable.

2.2 Machine-Code Analysis

Even though using high-level semantics for program verification may seem appealing, we advocate performing software verification at the machine-code level. In the case where we only have a machine-code program (e.g., an executable downloaded from the Internet), direct source-level analysis is impossible. Even when high-level source code is available for analysis, the implementation of the programming language needs to be trusted (or verified); for example, the memory semantics being offered by the programming language are eventually provided by the underlying memory system. Moreover, analysis of high-level programs does not account for the possibility of bugs introduced by compilers, thereby decreasing confidence that properties proved during analysis will hold at run-time. Even with the advent of practical and verified mainstream compilers [49], assurances that a program does not access unauthorized memory regions cannot be obtained unless low-level operating system code, on which programs rely for various services [50–53], is also analyzed; often, such low-level OS code is written in assembly language. Therefore, it is prudent to analyze machine code for modern commercially available processors. A big benefit of machine-code analysis tools is their universal applicability; they can be used to verify all programs that can compile down to the supported hardware platform. Also, similar to the CLI stack, machine-code analysis can be used to prove the programming language implementation on a given processor correct; software verification can then proceed at the programming language level.

Our focus is on the 64-bit mode of the x86 ISA. There are two main reasons for choosing the x86 ISA over other available ISAs.

1. Since x86 is the dominant processor architecture for non-embedded devices, a simulation and verification framework based on the x86 ISA will find immediate practical application. Also, most modern non-embedded devices run in 64-bit mode.
2 The x86 ISA is one of the most complicated processor architectures, and hence, any success in this project will demonstrate that our formal methods technology is mature enough to handle real-world, industrial problems.

In spite of the benefits of machine-code analysis, formal verification of machine code has few practitioners [54–56], though it is steadily gaining ground. Information contained in the abstractions provided by high-level languages, like the layout or "shape" of data structures, is mostly lost at the machine-code level, which makes machine-code analysis detail-intensive and tedious. Work has been done to provably lift low-level verification to a higher-level reasoning process [57–59]. However, in this chapter, our focus is not on verification techniques themselves but on the engineering decisions involved in just constructing a reliable model of the x86 ISA to enable the application of those techniques.

2.3 Design Goals

Inspection of the behavior of machine code is often done by employing instruction-set simulators [60–62], which use sophisticated techniques to achieve high-speed simulation of machine instructions and are written in efficient programming languages like C and C++. Even though such simulators are commonly used as ISA reference models, they are not formal specifications of these ISAs and hence, do not directly provide an infrastructure for formal code analysis.

ISA models written in formal specification languages allow the direct application of formal reasoning tools. Verification using these formal models indeed increases confidence in the reliability of machine code, but this analysis can not be fully trusted until the model is known to be faithful to the processor. One way to increase confidence in the accuracy of a model is by performing co-simulations to validate it against a real machine. Co-simulation is the process of executing a machine program on the processor as well as on the model; if the state of the processor and the model are the same after every instruction, then the model is faithful to the processor for at least those instructions (and data). Unfortunately, formal specification languages do not usually provide an efficient execution environment. Model validation can also be done by performing extensive code reviews — an approach that is independent of the model's execution efficiency. However, code reviews are time and labor intensive. The reviewers would need to study the vendors' ISA manuals, which are long documents mostly consisting of prose [4, 63]. There is always the danger of making subjective judgments about the processor's behavior, especially if the processor has a complicated ISA and if some features are under-specified. Therefore, code reviews can supplement co-simulations, but not supplant them.

This comparison of ISA simulators and formal ISA specifications justifies the need to develop a formal ISA model whose execution speed allows efficient co-simulations for model validation. A validated model increases trust in the applicability of the results of formal verification. Another benefit of such a unified model is that it reduces the overhead associated with designing, developing, and maintaining two separate models — in particular, maintaining them consistently. ACL2 is an excellent fit for this task; it is both a mathematical logic and a programming language.

We considered the following factors over the course of engineering our model:

Accuracy No simplification in the semantics of the ISA may be done. For example, a machine model should be based on machine integers, not unbounded integers.

Efficiency The benefits of a unified model for simulation and formal verification come at a price: we must mitigate the trade-off between simulation and reasoning efficiency. If the model is defined using simple constructs, it enables easier reasoning but can offer poor execution performance. Definitions optimized for execution efficiency are typically more difficult to reason with because they may use an algorithm different from the "obvious" one. Abstraction techniques (e.g., those provided by ACL2 features like `mbe` and abstract stobjs [64]) provide leverage to build an efficient model for both reasoning and execution.

Usability In its role as a simulator, the model must come with dynamic instrumentation tools, similar to the GNU Debugger (GDB) and Intel's Pin Tool [65]. Such tools facilitate examination of the machine state during concrete simulations of programs, thereby helping in model and program debugging. In its role as a reasoning framework, the model must be accompanied with libraries that aid in machine-code analysis. For ease of use, the model is documented (available online [13]) — from the point of view of a simulator as well as a reasoning framework.

The complexity of the x86 ISA model will increase as more features are added to it, and this added complexity will inevitably make reasoning more involved. Balancing *verification effort* and *verification utility* is a highly pertinent issue. For example, users might not want to reason about an application program at the level of physical memory, i.e., taking into account address translations and access rights management. This is because it is customary for application programs not to have direct access to the system data structures.[3] The memory model seen by application programs is that of linear memory, which is an OS-constructed abstraction of the complicated underlying memory management mechanisms that are based on physical memory. Therefore, verification of application programs can be performed at the level of linear memory, if the OS routines that manage the linear memory abstraction can be either trusted or proved correct. However, system programs, like kernel routines, must necessarily have a view of physical memory since these programs can access/modify system data structures. In light of the above, the x86 ISA model should provide the option to deactivate some ISA features, enabling the user to do varying depths of analysis, depending on the kind of programs being considered for verification. Consequently, our x86 model has two modes of operation: the *programmer-level mode* for the analysis of application programs and the *system-level mode* for the analysis of system programs. Details about these modes are in Sect. 3.3.

Usability is further supported by the free availability of our model, under a BSD 3-Clause license, from the Github webpage of the ACL2 Community books [12]. This permissive license can facilitate the adoption of our model.

Scalability and Maintainability The model of the x86 ISA will be large and it will keep growing larger, due to complexity of the ISA and addition of new features. Thus, the maintainability of the model is an important factor influencing its design decisions; the two modes of operation of the model help in this regard.

[3]The access rights of an application program are ultimately governed by the OS, but it is extremely rare (and inadvisable) for an application program to have direct access to system resources.

3 Model Definition

Our ACL2-based formal and executable model [66] of the x86 ISA has a specification of all addressing modes, most user-level instructions (including floating point instructions), some system-level instructions, paging, and segmentation. The instructions that are unspecified, as of this writing, include those that access co-processor I/O address spaces, rarely used BCD instructions, cache-related instructions and virtual machine extensions.

We use the Intel manuals [4] as reference documents for model development. Ambiguities are resolved by cross-referencing AMD manuals [63], running tests on real processors, and consulting with x86 processor architects.

Our x86 ISA model provides an interpreter-style operational semantics [67], where the meaning of a machine-code program is given by a recursively-defined interpreter over the machine-code language's syntax and the processor's state. Broadly, there are four main aspects to a model defined using an interpreter-style operational semantics; we describe them using the x86 ISA.

1. **State**: Components of the x86 ISA state currently supported by our model are described in Table 1. Sections 3.1–3.3 describe how the state is defined and accessed in our x86 model.
2. **Instruction Semantic Functions**: Each machine instruction is specified by a function that defines its semantics. This semantic function takes an initial x86 state as input and returns an appropriately modified next state as output. Section 3.4 describes the instruction semantic functions in our model.
3. **Step and Run Functions**: A step function fetches, decodes, and executes a machine instruction by calling its associated instruction semantic function. A run function takes the number `n` of instructions to be executed and an initial x86 state. The run function function either takes `n` steps or terminates early if an unrecoverable error is encountered, returning an appropriately modified final x86 state in either case. Section 3.5 describes the step and run functions in our model.

The semantics of an x86 machine-code program are given by the composition of the semantics of its constituent instructions, and we can use ACL2 both to simulate concrete program runs and to reason about symbolic program runs.

Our x86 model is layered to facilitate ease of maintenance and scalability. We describe each layer below and discuss the reasons for our modeling decisions.

3.1 Concrete State

The x86 state is defined using an ACL2 data structure called a *concrete stobj* [68]. "Stobj" stands for "Single-Threaded Object"; such a structure is a mutable object with applicative semantics. Stobjs offer high execution performance by allowing destructive assignments while providing copy-on-write semantics for reasoning. Consider the `defstobj` event below that introduces a stobj called `foo`.

Table 1 Components of the x86 processor that our currently supported by our model

#	Component	Type and size	Comments
1	General-Purpose Registers	16 64-bit wide registers	`rax`, `rcx`, `r11`, etc.
2	Instruction Pointer	1 64-bit wide register	`rip` register
3	Flags Register	1 64-bit wide register	`rflags` register
4	Segment Registers	6 16-bit wide registers	`cs`, `ss`, etc.
5	Segmented Memory Management Registers	2 80-bit wide registers	Global and local descriptor registers, `gdtr` and `ldtr`
6	Interrupt and Task Management Registers	2 16-bit wide registers	Interrupt descriptor and task registers, `idtr` and `tr`
7	Control Registers	16 64-bit wide registers	`cr0` to `cr15`
8	Floating-Point Data Registers[a]	8 80-bit wide registers	`fp-data0` to `fp-data7`
9	Floating-Point Control Register	1 16-bit wide register	`fp-ctrl`
10	Floating-Point Status Register	1 16-bit wide register	`fp-status`
11	Floating-Point Tag Register	1 16-bit wide register	`fp-tag`
12	Floating-Point Last Instruction Pointer	1 48-bit wide register	`fp-last-inst`
13	Floating-Point Last Data Pointer	1 48-bit wide register	`fp-last-data`
14	Floating-Point Opcode	1 11-bit wide register	`fp-opcode`
15	XMM Registers	16 128-bit wide registers	`xmm0` to `xmm15`
16	MXCSR Control and Status Register	1 32-bit wide register	`mxcsr`
17	Machine-Specific Registers	6[b] 64-bit wide registers	
18	Byte-Addressable Memory	Models 2^{52} bytes[c]	Main memory

[a]The MMX registers are aliased to the low 64 bits of the FPU's data registers, as dictated by the ISA

[b]Intel defines a lot more than 6 model-specific registers. Our model currently supports 6 of them: `ia32_efer`, `ia32_fs_base`, `ia32_gs_base`, `ia32_kernel_gs_base`, `ia32_lstar`, `ia32_star`, `ia32_fmask`

[c]2^{52} bytes is the largest physical address space provided by modern x86 implementations

```
(defstobj foo
  (a :type (array (signed-byte 64) (2)) :initially 0)
  (b :type (unsigned-byte 48) :initially 10))
```

In logic, the stobj foo is simply a list of two elements; the first element (referenced by the field a) is a list of two elements, each of which is a signed integer of width 64 and has an initial value of 0, and the second element (referenced by the field b) is an unsigned integer of width 48 and has an initial value of 10. For execution, ACL2 enforces sequencing updates to foo by way of syntactic restrictions ensuring that only one instance of foo exists at all times, thereby allowing destructive assignments. Apart from introducing a stobj, a defstobj event also introduces some native functions: recognizers (functions that check whether the stobj and its constituent fields are of the right logical representation), a creator (a function that creates an initial logical representation of the stobj), accessors (functions that read a stobj field), and updaters (functions that write to a stobj field).

We have modeled components listed in Table 1 as fields in the x86 concrete stobj. For components like the segment registers, task register (tr), and local descriptor table register (ldtr), our model of the state also includes the hidden or shadow parts of these registers, as described in the processor manuals. Some fields in our x86 state are an artifact of the model itself — they do not exist on a real x86 processor. These fields store information about the model state (ms), mode of operation (programmer-level-mode), environment (env), seed for undefined values (undef), and information about the operating system (os-info). The ms field contains information about problems (if any) encountered during a program run. These problems include those arising due to features that are currently unimplemented in our model. The processor state is expected to be correct if ms is empty; otherwise, the run function terminates and processor execution is halted. Other fields will be discussed when relevant in the rest of this chapter.

Though the use of concrete stobjs to define the x86 state helps in achieving high simulation efficiency, we can get better performance by choosing the types of the stobj fields judiciously. The stobj fields should be defined in a way that avoids bignum creation when possible. One common technique we employ for this purpose is illustrated by our modeling of the general-purpose registers (GPRs), which are 64-bits wide on an x86 processor. On most contemporary 64-bit Lisp implementations, 64-bit numbers are of type bignum. The GPRs are defined as an array of 16 elements, each of which is a *signed* integer of width 64 bits instead of an *unsigned* 64-bit integer. This approach provides a substantial performance boost because big positive numbers can be stored as small negative numbers; e.g., bignum $2^{64} - 1$ can be represented by -1 when stored as a 64-bit signed integer. Similar optimizations have been made for other fields in the x86 state.

The byte-addressable memory deserves special mention. We support a memory of size 2^{52} bytes (i.e., 4096TB), the largest provided by modern x86 implementations. However, allocating 4096TB all at once is impractical, if not impossible. We have implemented a time- and space-efficient memory model [69] in order to keep the model's memory footprint manageable. Our memory model can be viewed as a flat array consisting of 128MB blocks; memory is allocated on demand in these blocks of 128MB instead of all at once. Three fields in the concrete stobj are used to implement this on-demand memory, `mem-table$c`, `mem-array-next-addr$c`, and `mem-array$c`. `mem-table$c` stores the 25-bit addresses of the 128MB blocks. `mem-array-next-addr$c` stores the 25-bit block address to be allocated next, and `mem-array$c` stores the bytes. Hence, the 52-bit physical address of a byte can be thought of as composed of two parts: the most significant 25 bits are the address of a block and the remaining 27 bits are the address of the offset within that block. The following pseudo-code describes how to compute `memArrayIndex`, the index into `mem-array$c` where the byte corresponding to a physical address `addr` is located; the notation `addr[to:from]` represents the slice of `addr` from bit position `to` to bit position `from`, inclusive of both indices, and `x86$c` represents the concrete x86 state.

```
blockAddr := mem-array(addr[51:27], x86$c)
if valid-p(blockAddr) then
          // Address located in a block previously allocated.
        memArrayIndex := (blockAddr <<  27) | addr[26:0]
else      // A new block needs to be allocated.
        addNewBlock(x86$c)
        memArrayIndex := (mem-array-next-addr << 27) | addr[26:0]
        mem-array-next-addr := mem-array-next-addr + 1
endif
```

With this notion of on-demand memory in our model, a well-formed x86 state needs to have a well-formed memory `good-memp`, i.e., the relationship among the three memory fields should give a correct model of a 4096TB byte-addressable memory. We have proved that this is an invariant of our model. Note that this is a property *about* our model — it assures us that our memory model, despite heavy optimizations, is indeed the one we intended to build.

We access and update a byte in the memory using functions called `mem$ci` and `!mem$ci`. These functions locate a byte using the relationship among the concrete state fields `mem-array$c`, `mem-array-next-addr$c`, and `mem-table$c` described above. We never access or update these three fields directly.

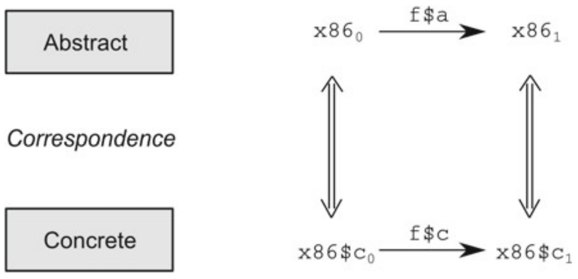
3.2 Abstract State

Though concrete stobjs enable reasoning and efficient simulation, using them has the following drawbacks in the context of our x86 model.
Issue (1) Large logical representation of the x86 state: Logically, the concrete x86 state is a list of lists and integers. Two of the fields that implement the memory model are extremely large linear lists; `mem-table$c` is a list of 2^{25} elements and `mem-array$c` is a list of 2^{28} elements (corresponding to an initial memory allocation of 256MB), which increases if more memory is requested. The size of these lists adversely affects reasoning efficiency — they have to be created in order

to symbolically simulate functions (in logic, using the bit-blasting tool GL mentioned above) that take the x86 state as input.
Issue (2) Expensive guard checking: The recognizer of a well-formed x86 state, say `x86$cp`, is the conjunction of the concrete stobj recognizer `x86$cp-pre` and the well-formedness of the memory model `good-memp`. Any function, say `g`, that takes the concrete x86 state as input would need to have `x86$cp` as a guard. However, `good-memp` is an expensive function because it has to check each element of all the three memory-related fields to ensure that a good relationship among them holds. Costly guard checking slows down `g` whenever it is executed on concrete data in ACL2. We use abstract stobjs [64] to overcome these issues. An abstract stobj may be viewed as an alternative representation of a *corresponding* concrete stobj. Figure 1 illustrates the idea behind abstract stobjs, which is in the spirit of bisimulation. Let `x86` be an abstract stobj, `x86$c` a corresponding concrete stobj, and a function `f` be associated with the abstract and concrete functions `f$a` and `f$c` that update the abstract and concrete stobj, respectively. Then, $\texttt{x86\$c}_1$ corresponds to $\texttt{x86}_1$ provided that:

- `f$a` maps instance $\texttt{x86}_0$ of `x86` to $\texttt{x86}_1$.
- `f$c` maps instance $\texttt{x86\$c}_0$ of `x86$c` to $\texttt{x86\$c}_1$.
- The correspondence predicate holds for $\texttt{x86\$c}_0$ and $\texttt{x86}_0$.

Admitting an abstract stobj requires explicitly defining the correspondence predicate and discharging proof obligations that establish the correspondence of the concrete and abstract state at all times. Some examples of these are: the so-called *correspondence* theorems, that establish that the initial concrete and abstract states correspond, the concrete and abstract accessor functions for the same field produce the same value when applied to corresponding states, and the concrete and abstract updater functions for the same field produce corresponding states when applied to corresponding states; and the *preservation* theorems, which state that the recognizer always holds for the abstract state — we prove that the recognizer holds for the initial state, and it is preserved by every well-guarded updater function. For details about these theorems, see the documentation in the ACL2 manual [70]. Once an abstract stobj has been admitted, any native function accessing or updating it reduces to a simple function associated with the abstract state during reasoning and to an efficient function associated with the concrete state during execution.



$$\begin{tikzcd} \texttt{x86}_0 \arrow[r, "\texttt{f\$a}"] \arrow[d, Leftrightarrow] & \texttt{x86}_1 \arrow[d, Leftrightarrow] \\ \texttt{x86\$c}_0 \arrow[r, "\texttt{f\$c}"] & \texttt{x86\$c}_1 \end{tikzcd}$$

Fig. 1 Abstract and concrete x86 states

We define the x86 abstract state corresponding to the concrete state by using records [71] to model the concrete array fields. A record is a sparse association list — it is a finite normalized structure that maps keys to non-default values. Memory is represented by a single record in the abstract state; contrast this with three fields being used to represent memory in the concrete state. Functions `memi` and `!memi` invoke access and update operations (respectively) on records when reasoning and invoke `mem$ci` and `!mem$ci` (respectively) during execution. The initial representation of the abstract memory is empty, as opposed to a large linear list of zeroes for the concrete memory fields. The abstract memory contains only the values that have been written explicitly to the memory. This results in a smaller x86 state that is more amenable to reasoning, thereby solving Issue (1). The recognizer for the abstract x86 state always evaluates to true during execution, as justified by the recognizer preservation theorems. This solves Issue (2).

A benefit of using abstract stobjs is that the concrete state and its associated functions can be optimized for execution efficiency without affecting the abstract layer and other definitions of the model built on top of it, as long as the correspondence relation is maintained. Thus, maintenance of the model becomes tractable and a trade-off between reasoning and execution efficiency is avoided.

Theorems about Reads and Writes to the x86 State A program's behavior can be described by the effects it has on machine state. Given an initial state, the final state may be described as a nest of updates made in program order to the initial state. In order to reason about the behavior of a program, we need to develop lemmas to read from, write to, and re-arrange these nests of updates.

- *Read-over-Write Theorems*: There are two types of Read-Over-Write theorems. The first describes the independence or non-interference of different components of the x86 state. One example is proving that an update made to a specific register does not modify the value of any other component of the x86 state. The second asserts that reading a component after it is modified returns the value that was written to it during the modification.
- *Write-over-Write Theorems*: Like the Read-Over-Write theorems, there are two types of Write-over-Write theorems. The first asserts that independent writes to the x86 state can commute safely. The second asserts that if consecutive writes are made to a component, the most recent write is the only visible write.
- *State Preservation Theorems*: These theorems assert that writing a valid value to a component in a valid x86 state returns a modified x86 state that is still valid.

We need these theorems for every component of the x86 state, but the number of theorems required is quadratic in the number of components [72]. Adding components to the x86 state to support more features would entail proving more of these theorems. Apart from being tedious, proving such a large number of theorems can slow down the theorem prover's rewriter. Instead, we define interface functions `xr` and `xw` to read from and write to the x86 state, respectively. The functions `xr` and `xw` take field names as input; they branch on the field name and call the corresponding native accessor or updater function. Now these theorems have to be in terms of just these two functions — we have just five theorems to manage.

Though this simplifies reasoning, simulation efficiency suffers because we incur an extra function call every time the x86 state needs to be accessed or updated; every definition in the model will use these functions. Inlining `xr` and `xw` can avoid the cost of the function call, but because these functions contain a big `case` statement, this would increase the code and memory footprint of our model.

We solve this problem by using ACL2's `mbe` feature. For each component of the x86 state, we define new accessor and updater functions whose body is an `mbe`, where the `:logic` parts call `xr` and `xw` (respectively) and the `:exec` parts call the native accessor and updater functions. We then keep these new functions enabled for reasoning — when these functions are called, reasoning will be done in terms of `xr` and `xw`. We inline these functions so that execution will use the native accessors and updaters, which in turn will use the efficient concrete stobj functions. Thus, these new functions are now the top-level interfaces to the x86 state. There still remains the (small) issue of having to define two functions (a new accessor and updater) for every field added to the state. We use Lisp's (and ACL2's) ability to treat code as data and automatically generate these functions from the stobj definition.

With this mechanism in place, whenever a model developer adds a new field to the x86 state, he does not have to worry about manually proving read-over-write, write-over-write, and state-preservation theorems, and adding new accessor and updater functions that have an `mbe` in their body. This makes the model maintainable, which is critical for the longevity of a model as large as that of the x86 ISA.

3.3 Modes of Operation: Interface to the x86 State

As mentioned in Sect. 2.3, our x86 model offers two modes of operation: the programmer-level mode and the system-level mode. These modes can be thought of as interfaces to the x86 state; the programmer-level mode provides the same interface to the x86 state as is provided by the operating system to application programs and the system-level mode provides the same interface to programs as is provided by the processor. Thus, these modes enable simulation and reasoning about x86 machine-code programs either from the point of view of an application programmer or a kernel developer. A field in our x86 state, `programmer-level-mode`, is set to `t` when operation in the programmer-level mode is required and to `nil` otherwise.

3.3.1 Programmer-Level Mode

This mode allows the analysis of an application program while assuming that memory management, I/O operations, and other services are provided reliably by the underlying OS. We discuss some aspects of this mode below.

Memory Model In the 64-bit mode of an x86 processor, application programs use 64-bit linear addresses. The x86 ISA defines *canonical addresses* as those which have bits 63 through 48 set to all zeros or all ones; if a non-canonical address is encountered, then a general protection exception or a stack fault occurs, depending on the context. This means that the canonical linear address ranges are 0 to $2^{47} - 1$ and $2^{64} - 2^{47}$ to $2^{64} - 1$, i.e., 2^{48} bytes of linear address space is the total available space. Note that the second canonical address range would result in addresses that are of type bignum in Lisp implementations. As discussed in Sect. 2.1.1, it is advantageous to avoid bignum creation when possible. In this case, we choose to represent linear addresses as signed integers of width 48 — these will always be fixnum values on 64-bit Lisp implementations. The linear memory range 0 to $2^{47} - 1$ is represented as is, and the other range $2^{64} - 2^{47}$ to $2^{64} - 1$ is represented by -2^{47} to -1.

In the programmer-level mode, our byte-addressable memory model discussed in Sect. 3.1 serves as the linear memory space and only up to 2^{48} bytes of the memory model can be accessed — every non-canonical linear access is an error, in accordance with the ISA. The functions `memi` and `!memi` are not used to read from or write to the memory in this mode (because they can access the entire 2^{52} bytes of available memory). Linear memory accessor and updater functions that restrict accesses to 2^{48} bytes are defined and used instead. The guards of these linear memory functions require the address to be a 48-bit signed integer, which is converted to a 48-bit unsigned integer when indexing into our memory model (for example, -1, which represents the canonical address $2^{64} - 1$, is converted to $2^{48} - 1$).

The programmer-level mode provides some support for *user-level* 64-bit segmentation, even though paging is unavailable in this mode. Segment descriptor tables (GDT and LDT) are unavailable in this mode, but if a logical address refers to the `FS` or `GS` segments (which are the only segments that can have non-zero bases in the 64-bit mode), the segment base addresses are obtained from the machine-specific registers `ia32_fs_base` or `ia32_gs_base`, respectively. It is the responsibility of system software to load these registers, and hence, they are assumed to contain appropriate values in this mode, thereby providing an environment similar to that provided by an OS to application programs.

System Calls and Non-determinism Application programs rely on various services provided by the OS. These services are usually provided by system calls. Some of these system call services are non-deterministic from the point of view of a programmer — different runs can yield different results on the same machine. For example, the `open` system call might open a file in one execution, but result in an error in another execution if the file has been deleted. The `syscall` instruction is used by application programs to call system procedures at a higher privilege level in order to request services from the underlying OS; the value in register `rax` determines which system call is to be executed. The `sysret` instruction is used by system programs to return control to the application-level program that requested the service. System call implementations differ in different OSes; a small example is that the `read` system call has the number 0 on Linux and 3 on Darwin.

The x86 instructions `syscall` and `sysret` are special in this mode. The semantic function of `syscall` is extended to provide the semantics of system calls like `read` and `write`, and the instruction `sysret` is unavailable for use. A field in our x86 state, `os-info`, also needs to be initialized appropriately (either `:linux` or `:darwin`, which are the two OSes we currently support).

For system call simulations in the programmer-level mode, we use the return values of functions that invoke the system calls by interacting directly with the underlying OS. That is, simulation of all instructions except `syscall` occurs within ACL2; for `syscall`, we determine which system call is requested and then escape out of ACL2 to get the real results by executing that system call on the underlying system. Obviously, these functions are *impure*: logically, they are not functions because they can return different values for the same input arguments.

For reasoning about system calls, we incorporate an environment field `env` into the x86 state to represent the part of the external world that affects or is affected by system calls. The `env` field contains a specification of a subset of the *file system* as well as an *oracle* field. The oracle specifies the result of any action that is non-deterministic from the point of view of the programmer; it provides information that, though a part of the real environment, cannot be inferred from our model of the file system. An example of such information is the file descriptor of a file to be opened — an OS assigns the file descriptor depending on the number of files already opened for a particular process at the time the `open` system call is made. The oracle contains a map that associates linear addresses to a list of arbitrary values; whenever a value is needed from the oracle, the current instruction pointer is located in the map, and the first value in the corresponding list is popped off for use and removed from the list. If the program control goes to the same instruction later and the oracle is consulted, again the first value (previously the second value) in the list is popped off and then removed. Thus, to reason about a system call's effects, it is the responsibility of the user to initialize the `env` field appropriately so that it can be consulted when needed.

For example, suppose that a program makes an `open` system call to open a file, then a `write` system call to write to that file, and finally a `close` system call to close it. The `env` field's file system fields would need to be initialized to contain the file information — its name and descriptor — and the oracle field would need to associate the linear address of the first `syscall` instruction (i.e., `open`) with the value of a descriptor. Broadly speaking, reasoning about this program will involve reasoning about the following behavior: the first `syscall` instruction will pop off the descriptor from the oracle field, associate it with a file name, and change its mode as requested; the second `syscall` will use the descriptor to locate the file and write the requested number of bytes from a specified buffer to the file; and, the third `syscall` will remove the association between the file descriptor and its name. Our specifications of these system calls include accounting for error conditions too.

The contents of `env` can be symbolic. For example, to verify a program `wc` that counts the number of words in a given file, the contents of the file can be specified as an arbitrary string in the `env` field's file system fields. Of course, it is also possible to reason about concrete elements in the environment, e.g., `wc` on a concrete file, by simply initializing the `env` field with these elements. The `env` field is invisible and inaccessible when the model is used as a simulator. System call simulation and reasoning use different mechanisms and do not interfere with each other. However, the following meta-connection exists between them in our

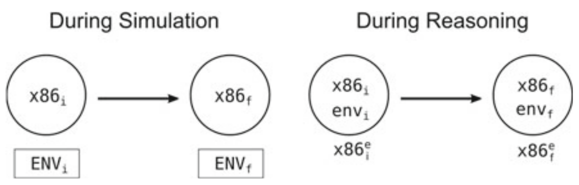


Fig. 2 Environment during simulation and reasoning

x86 model; see Fig. 2. Let $\texttt{x86}_\text{i}$ be an x86 state. Suppose, during simulations, the evaluation of $\texttt{run}(\texttt{x86}_\text{i})$ returns $\texttt{x86}_\text{f}$ and updates the real environment from $\texttt{ENV}_\text{i}$ to $\texttt{ENV}_\text{f}$. Then, the following is true during reasoning: if $\texttt{env}_\text{i}$ corresponds to $\texttt{ENV}_\text{i}$, and $\texttt{x86}^\text{e}_\text{i}$ refers to $\texttt{x86}_\text{i}$ augmented with $\texttt{env}_\text{i}$, then the evaluation of $\texttt{run}(\texttt{x86}^\text{e}_\text{i})$ during reasoning produces $\texttt{x86}^\text{e}_\text{f}$, which is $\texttt{x86}_\text{f}$ augmented with $\texttt{env}_\text{f}$, for some $\texttt{env}_\text{f}$ corresponding to $\texttt{ENV}_\text{f}$. To ensure that this connection holds for our model in the programmer-level mode, we perform co-simulations by comparing concrete program simulations to corresponding evaluations done inside ACL2, that is, using pure logical functions. More details can be found in our FMCAD'14 paper [73].

We use `env` to characterize all non-deterministic behavior, not just system calls in programmer-level mode. For example, the `RDRAND` instruction (read random number) is available both in the programmer-level and system-level modes, and for reasoning, it obtains a random value from the oracle in `env`.

3.3.2 System-Level Mode

The system-level mode is intended to be used to simulate and verify software that has supervisor privileges and access to system state, though it can also be used to verify application programs. We discuss some components of this mode below.

Physical Memory Our model includes 2^{52} bytes of byte-addressable memory; this is the largest physical address space provided by modern x86 implementations. The memory read and write functions `memi` and `!memi` pertain to physical memory.

Paging Paging is used for address translation and access rights management. In 64-bit mode, physical memory cannot be directly accessed by either system or application software. In fact, switching to the 64-bit mode requires paging to be turned on first. The system-level mode supports IA-32e paging for all configurations (4KB, 2MB, and 1GB pages) of paging data-structures, including predicates that check whether paging structure entries are valid. A linear memory address, coupled with information necessary to check access rights like the origin[4] of the address, is used as input to the paging mechanism. A linear memory access will report a failure in our model if a page-fault exception is encountered — exceptions are currently unsupported by our model. We have specified a "page walk" by defining it in terms of traversals of

[4]Determining the origin of a linear memory address constitutes answering questions like: is the linear memory reference on behalf of a read, a write, or an instruction fetch and decode? What is the privilege level, as determined by the segmentation data-structures, of this linear memory reference?.

each of the hierarchical paging structures. These traversal functions also perform on-the-fly updates to the accessed and dirty flags, as specified by the x86 ISA. Thus, in this mode, every read or write to a linear memory address goes through paging: permissions are checked to see if the memory access is allowed and the linear address is translated to its corresponding physical address.

Figure 3 represents the paging data-structures with a 4 K page configuration — we assume that the address translation does not result in a page-fault exception. The paging specification functions clearly require many slicing operations, i.e., getting and setting some ranges of bits in a number. These functions have been optimized for execution efficiency by using type declarations and guards. We use pre-existing ACL2 libraries [74] to aid in this task.

Segmentation In IA-32e mode, much of the functionality of segmentation is disabled. Segment limit checks and null segment selector checks are not performed, but descriptor table limit checks, type checks, and privilege-level checks are still carried

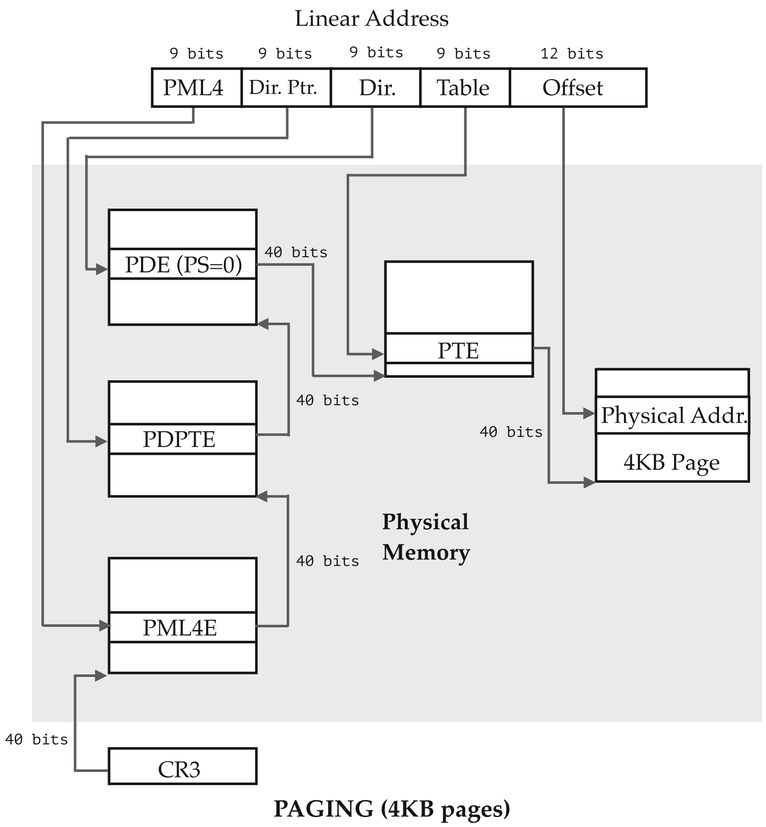
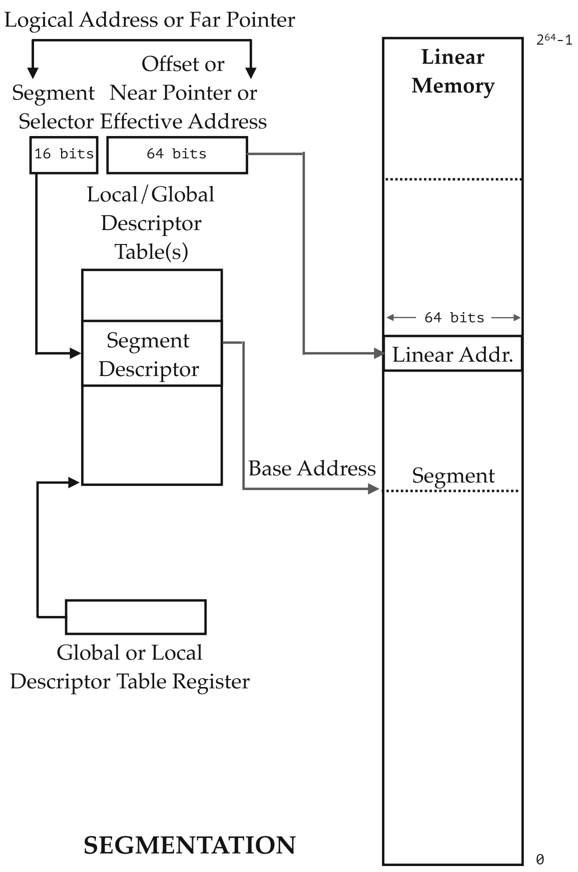


Fig. 3 IA-32e paging with 4KB page configuration

Fig. 4 IA-32e segmentation


Logical Address or Far Pointer
Segment Selector
Offset or Near Pointer or Effective Address
16 bits
64 bits
Local/Global Descriptor Table(s)
Segment Descriptor
Base Address
Global or Local Descriptor Table Register
Linear Memory
2^{64}-1
64 bits
Linear Addr.
Segment
0
SEGMENTATION

out. The segment base for all segments except FS and GS is treated as zero; for FS and GS, logical address to linear address translation is depicted in Fig. 4.

Segment descriptors are data structures inside the descriptor tables that contain information about a segment — its size, location, access control and status information. We have defined predicates that recognize valid segment descriptors. We have also specified 64-bit segmentation-related privileged instructions like `LGDT` (load global descriptor table register) and `LLDT` (load local descriptor table register); these instructions check the descriptor table limits and the validity of the segment selector and descriptor, and then load the `gdtr` or `ldtr` register appropriately.

Segmentation plays an important role in fast system calls, i.e., `syscall` and `sysret` instructions, because these instructions involve changes in privilege levels. Unlike in the programmer-level mode, the `syscall` instruction's specification in this mode is exactly what is specified by the x86 ISA. The `sysret` instruction is also available for use in this mode. See Fig. 5 for an illustration. These instructions

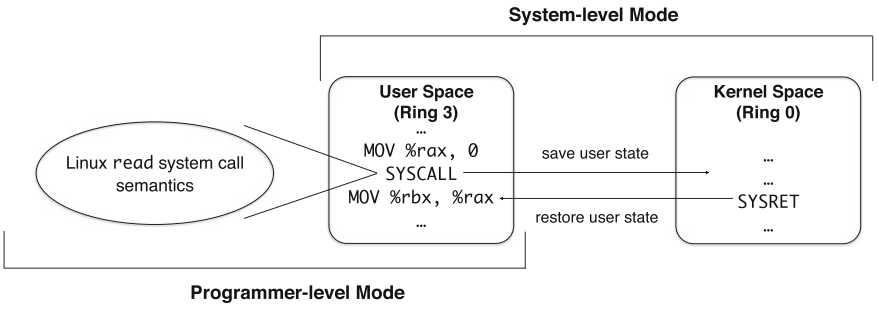


Fig. 5 Linux `read` system call in the two modes of operation of the x86 model

access system state like the machine-specific registers and code segment descriptors that describe where control is to be transferred if all privilege checks succeed.

3.3.3 Implications for Reasoning

Of all the differences from the programmer-level to system-level mode, the memory model adds the most complexity. The pseudocode for the top-level function for accessing a byte, `rm08` (which stands for "read memory — 8 bits"), is as follows. Note that `LaToPa` is a large, complex function does address translation via paging.

```
rm08(linAddr, permissions, x86):
if non-canonical-address(linAddr) then
    raiseError(x86)
else
    if programmer-level-mode(x86) then
            memi(linAddr, x86)
    else
            cpl := get-rpl-from-cs-segment(x86)
            [err, phyAddr, x86]:= LaToPa(linAddr, permissions, cpl, x86)
            if err then
               raiseError(x86)
            else
               byte := memi(phyAddr, x86)
               return (byte, x86)
            endif
    endif
endif
```

For convenience, we have similar top-level functions for accessing words, double-words, quadwords, and octawords. This means that for reasoning, we would need the usual read-over-write and write-over-write lemmas for each of these top-level functions in both the modes. Similar to our discussion about `xr` and `xw` in Sect. 3.2, we avoid the need for proving this large number of theorems using ACL2's `mbe` feature; logically, all these functions are equal to a call of another function called `rb` (read bytes). The function `rb` takes a list of linear addresses, permissions, and the x86 state as input and if no error is encountered, returns a list of bytes at the input addresses and the x86 state. For the system-level mode, this equality to `rb` makes it possible to reason at the level of linear address space, which is akin to system programming in 64-bit mode. However, this equality is true only when the paging data-structures are set up correctly — e.g., no page fault error is encountered during `LaToPa`. This is essentially the same as stating that the programmer-level mode is an abstraction of the system-level mode as far as the memory is concerned, if the system data structures have been set up correctly.

We have adopted a similar strategy for reasoning about memory writes, where every write is logically equal to a call of `wb` (write bytes). Another benefit of using `rb` and `wb` is that all memory accesses are reduced to reasoning about list operations (like subset) as opposed to tedious arithmetic involving address ranges. Using lists in `rb` and `wb` does not affect the simulation efficiency of the model because these functions are used only during reasoning; for simulation, efficient stobj code is used.

3.4 Instruction Semantic Functions

Functions like `rm08` (see Sect. 3.3.3 above) abstract away the details of accessing the x86 state, irrespective of the mode of operation. This makes specifying the behavior of instructions relatively straightforward.

The instruction fetch, decode, and execute function (described in Sect. 3.5) passes on the decoded parts of the instruction to the appropriate instruction semantic functions, which use them to determine the locations of the operands (if any) and then fetches them from the x86 state. These functions then call an *operation specification function* — e.g., the `ADD` operation's specification function takes two operands and the flags as input and returns the result and the output flags. The instruction semantic function then writes the outputs from this specification function to the appropriate parts in the x86 state. A benefit of this approach is that instructions which fetch inputs and write outputs in the same manner can share the same semantic function; the actual computation is done by calling the correct operation specification function inside the semantic function. For example, the following instructions (and opcodes) have a single semantic function but different operation specification functions: `ADD` (`0x00`, `0x01`), `OR` (`0x08`, `0x09`), `ADC` (`0x10`, `0x11`), `SBB` (`0x18`, `0x19`), `AND` (`0x20`, `0x21`), `SUB` (`0x28`, `0x29`), `XOR` (`0x30`, `0x31`), `CMP` (`0x38`, `0x39`), and `TEST` (`0x84`, `0x85`). Again, this facilitates maintainability.

Characterizing Undefined Behavior Some x86 instructions leave certain parts of the machine state undefined. For example, according to the Intel manuals, the carry, parity, auxiliary, zero, sign, and overflow flags are all undefined after an unsigned divide instruction (`DIV`) is executed. A property of an "undefined value" is that an equality test with another value, either defined or undefined, should be indeterminate. This also implies that every undefined value should be *unique*, in the sense that any two such values are independent of each other. An illustrative example of why *indeterminateness* and *uniqueness* of undefined values is important is as follows: if a program contains a DIV instruction followed by a JGE instruction (which transfers control to a given address if the sign flag and overflow flag are equal, otherwise the control goes to the instruction following JGE), our analysis should not be able to determine whether the jump occurred or not. This alerts us to unreliable, if not dangerous, behavior that a program may exhibit. Thus, the specification of instructions like DIV should include such undefined behavior.

One way to model undefined behavior is to use 4-valued logic, where the unknown X meets our criteria. However, an issue with using 4-valued logic in our model is that we would need to use at least two bits to represent every bit in the processor state, thereby doubling the memory footprint of our model. Also, 4-valued logic is more complicated to reason with because X has a tainting effect. The usual 2-valued logic equal function cannot be used in cases where either or both of its input values can be X, in which case the result should also be X.

Instead, we model undefined behavior by defining a constrained function (see Sect. 2.1.1) called create-undef to provide indeterminateness and by adding a field called undef to our x86 state to provide uniqueness. Create-undef has an input and output arity of one; the only thing we know about the output is that it is an unsigned integer, irrespective of the input. Note that if the input to two calls of create-undef are different, then we can not determine if they are equal. The undef field contains an unsigned integer that is incremented every time an undefined value is required; thus, we get a unique value. We put undef and create-undef together by wrapping them up in another function called undef-read, which takes an x86 state as input, and returns a unique undefined value and a modified x86 state where undef is one more than what it was in the input x86 state. We ensure that undef-read is the only function that can modify the undef field by making the undef updater function, update-undef, untouchable (see Sect. 2.1.1 for details). Using undef-read will force us to reason about all the possible behaviors if a program relies on undefined values.

```
undef-read(x86):
        undef-seed        := undef(x86)
        new-undefined-val := create-undef(undef-seed)
        new-x86           := update-undef(1 + undef-seed)
        return(new-undefined-val, new-x86)
```

Characterizing undefined behavior is a bit different for concrete program simulations. Revisiting our DIV and JGE example, if that program runs on a real x86 machine, it would do exactly one of the following: jump to the given address or give control to the instruction following JGE. A simulator should behave in the same way as a real x86 processor; we do not have the luxury of accounting for all possible behaviors here. One way to solve this problem is to perform *native execution*. If our model is being used as a simulator on an x86 machine, then for every instruction that leaves any component undefined, we can escape from ACL2, re-run the same instruction on the real x86 machine, gather the values of the components that are supposed to be undefined, and pass those values up to the x86 model. These values can then be plugged into the corresponding fields in the x86 state in our model. This would require a trust tag (see Sect. 2.1.1) when the model is in use as a simulator. However, switching environments can significantly slow down the simulation speed of our x86 model. Instead, when our model is used as a simulator, we chose simply to assign suitable concrete values to the x86 components that are supposed to be undefined by an instruction. We also log the value of the instruction pointer when this occurs so that we have a record of which instructions were simulated that concretized any undefined behavior in this manner. The concrete value is chosen based on tests performed on a real x86 machine that indicate the most probable value for that instruction. This optimization is an example of balancing efficiency and accuracy; the simulator is accurate in that it captures a behavior that is indeed possible and it is efficient because the entire simulation takes place in ACL2/Lisp.

Note that the principle behind the oracle in the `env` field (see Sect. 3.3.1) is quite similar to that of the `undef` field. We can imagine discarding the `undef` field completely and using the oracle field in `env` to provide undefined values. However, the `undef` and `env` fields are used in different situations. For reasoning about programs that involve commonly occurring undefined events (like flag computations), using the oracle field can be tedious because it has to be initialized appropriately, i.e., a list of values has to be associated with an address which contains an instruction that involves any undefined behavior. Imagine doing that for all the undefined flags whenever `DIV` is executed. The `undef` field does not need to be initialized. The initialization of the oracle field provides a way of tracking any computation that relies on the external environment. Such computations do not happen often, and when they do, this initialization specifies exactly what we expect from the environment. In the case of undefined values, we do not have any such expectations — all we want is an infinite pool of unique and indeterminate values.

3.5 Step and Run Functions

The step function of our x86 ISA model fetches, decodes, and executes a machine instruction. It takes an x86 state as input, fetches the instruction from the memory at the location pointed to by the instruction pointer `rip`, and then dispatches control to the appropriate instruction semantic function. The x86 ISA has variable-length instructions, as depicted in Fig. 6 — the maximum length of a legal x86 instruction is

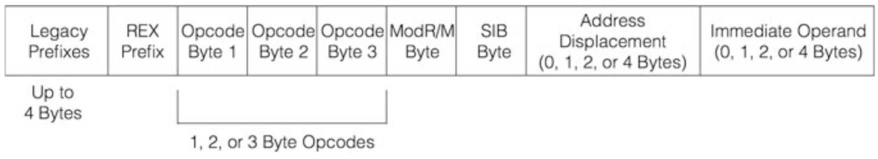


Fig. 6 x86 instruction format (64-bit mode)

15 bytes. Since most instructions are smaller than this limit, our step function lazily fetches one byte of an instruction at a time instead of fetching 15 bytes at once. Just enough bytes are fetched and decoded to determine the instruction opcode (so that the right instruction semantic function can be called), and the sizes and location of the operands. Decoding a byte of the instruction gives information about the nature of the next byte. E.g., if the first byte is a one-byte opcode, then the ISA tells us if that opcode requires a ModR/M byte. If it does not, then we dispatch control to the proper instruction semantic function, which, if needed, fetches the address displacement or immediate data encoded in the instruction or the operands from the x86 state. If the one-byte opcode requires a ModR/M byte, then the byte following the opcode is the ModR/M byte, which gives us yet more information about whether a SIB byte is expected, and so on.

The run function is the top-level specification function of our x86 ISA model.

```
x86-run(n, x86):
        if (ms x86) || (n == 0) then
           // Problem indicated by the model state field or
           // no more instructions left to be run
           x86
        else
           x86-run(n - 1, x86-fetch-decode-execute(x86))
        endif
```

4 Model Validation

A high simulation speed facilitates efficient co-simulations, which increase trust in the accuracy of the model, as discussed in Sect. 2.3. An added benefit of having two modes of operation of the x86 model is an increased simulation speed in the programmer-level mode, where much of the x86 state is abstracted away. For example, in the system-level mode, every access to the memory requires yet more accesses to the hierarchical data structures in the physical memory for memory management, whereas a memory access in the programmer-level mode is a direct operation.

The simulation speed of our model is ∼3.3 million instructions/second in the programmer-level mode and ∼330,000 instructions/second in the system-level mode.[5] We have successfully simulated a SAT solver,[6] which produced exactly the same effects as the real machine on the model's memory and registers.

Figure 7 illustrates our framework for performing co-simulations for model validation. Given a machine-code program in an executable format, we first determine if it can be simulated on our model. We then parse the executable file using our ACL2-based parser that determines, among other things, where the program is to be placed into the memory of our x86 state. Our ACL2-based loader loads the program

[5]This was measured on a machine with Intel Xeon E31280 CPU @ 3.50 GHz with 32GB RAM.

[6]This solver, developed by Marijn Heule, has performance comparable to state-of-the-art solvers.

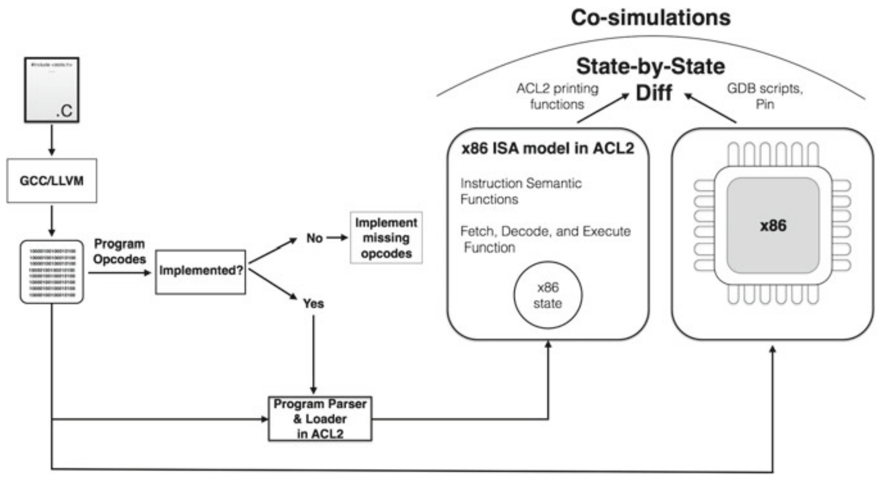


Fig. 7 Model validation via co-simulations

into the x86 state appropriately. The parser and loader are discussed in Sect. 4.1. We then execute the program on our model as well as on the real machine, taking care to initialize the state in our model appropriately (see Sect. 4.1.1 for details). The states of both the model and the machine are then logged in the same manner, i.e., either after every instruction or at any other granularity using instrumentation tools on the real machine and their equivalent on our model (Sect. 4.2). Any differences found between the model and the real machine are logged for later examination.

4.1 Machine Program Parser and Loader

We chose to write our machine program parser and loader in ACL2 itself. Writing our tools in ACL2 leaves open the possibility of formally reasoning about them. Also, we find it more convenient to debug and maintain ACL2 code — the books containing these tools are regularly [75] certified by ACL2.

Our parser and loader support machine-code programs in either ELF [76] format for Unix systems or Mach-O [77] format for Darwin systems. The parser reads the executable file, checks whether it is well-formed, and uses its headers to retrieve the various sections (e.g., text, data) in the file, along with information like their location in the memory. The loader takes this output from the parser and an initial x86 state as input, and places the sections in the right locations in the model's memory.

4.1.1 x86 State Initialization

For co-simulations, the initial state of the model should match the initial state of the real machine. The parser and loader only initialize the memory in the x86 state of our model. For convenience, we provide a function called `init-x86-state` to initialize all the components of the x86 state at once. This function also takes in an address, which, if encountered as an instruction pointer, halts the simulation of the program. Thus, simulating only a part of the program is also possible.

If simulation is being done in the system-level mode of the model, we provide a default paging data-structure configuration (identity-mapped 1GB pages). Of course, users are free to load their own system data structure configurations as well.

4.2 Instrumentation

For more control over the granularity at which the states of the model and a real machine are compared during co-simulations, the program must be instrumented in the same manner on the model and the machine. We use tools like the GNU Debugger and Intel's Pin [65] to instrument programs on a real machine. We have developed our own ACL2-based program instrumentation tools to provide similar capabilities for our x86 model. Our instrumentation tools are very flexible; e.g., we can trace every read and/or write to the x86 state (including the memory) or only trace those reads and writes that meet certain criteria. We can step through a program instruction by instruction, similar to the `nexti` and `stepi` commands on GDB, we can execute a specified number of instructions, and we can set breakpoints at arbitrary points in the program (e.g., when `rax` = 4 or when `rip` = 0x400952, etc.).

Apart from aiding in co-simulations, our instrumentation tools help in program comprehension. Our model is a programmable simulator — it provides a safe environment to simulate the program and examine its effects on the machine state.

5 Reasoning About x86 Machine-Code Programs

This chapter is primarily about the development and validation of our x86 ISA model, but because optimizing reasoning efficiency affected our modeling decisions, in this section we briefly discuss x86 machine-code verification using our model.

We have developed general libraries to reason about x86 machine code, both in the programmer-level mode and the system-level mode. An example of such a library is one which aids in reasoning about the non-interference/overlap of memory regions. The functions `rb` and `wb` (see Sect. 3.3.3) operate in terms of lists of addresses and bytes, making non-interference and overlap expressible in terms of notions like disjoint and subset. These libraries are mature enough to reason about non-trivial properties of a given program automatically, e.g., independence of the stack and heap from the program and data. We also have a library to reason about traversals of and updates to paging data-structures. We have formulated predicates that recognize a valid paging structure entry, and proved that if these predicates hold on a given entry, walking that entry will not result in a page-fault exception. Regardless of on-the-fly updates to these data structures during traversals (accessed and dirty bits), we have proved that repeated walks of valid entries do not modify the address mapping. Such lemmas enable reasoning at the level of linear memory reads and writes in the system-level mode. This is important because in IA-32e mode, even system software cannot access physical memory directly — linear memory is the abstraction that must be used.

We can automatically verify snippets of straight-line machine code using a BDD or SAT-based bit-blasting proof engine in ACL2 called GL [16, 78]. Providing the specification is the primary user requirement in order to prove theorems using this framework. If a conjecture fails, GL can compute counterexamples. Using this technique, we have verified a complicated population count (bit-count) program [79], and we detected a bug in an incorrect version [80]. Such an automated symbolic execution technique for reasoning about straight-line code can facilitate compositional verification, thereby reducing the cost of reasoning about larger programs.

A verification effort of note in the programmer-level mode is that of a word-count program [73], which computes the number of characters, words, and lines in a stream read in from the user using the `read` system call. We wrote a trio of simple ACL2 specification functions that compute these counts of a string. The final theorem asserts that the values returned by these three specification functions on standard input are found in the expected memory locations of the final x86 state, which is obtained by symbolically running the word-count program on our x86 model. Using our lemma libraries, we automated the proof of disjointness of the word-count program and its stack in every execution. We proved that irrespective of the size of the input file, this program always uses a fixed amount of memory on the stack to compute and store the counts. Another proof that was discharged automatically was that the word-count program does not modify unintended regions of memory, i.e., the only writes that occur during the program's execution are to the stack and the rest of the memory is the same as it was before the execution. Our lemma libraries reduced the manual effort required for machine-code verification of this program substantially, as demonstrated by some empirical evidence. The lines of ACL2 needed to verify the program without our libraries were appropriately 20 K, but with the libraries were appropriately 8 K. Even though 8 K lines may still seem inordinate, it should be noted that more than half of these 8 K lines were generated by ACL2 in response to requests to simplify certain symbolic expressions related to the execution of multiple instructions. These simplified expressions are large because a typical instruction makes many updates to the x86 state. The effort with 20 K lines also had around 4 K lines resulting from simplification of symbolic expressions.

In the future, we hope to use recently developed ACL2 tools like Codewalker [58] and Stateman [81] to increase automation. Codewalker is a machine-code decompilation tool — it automatically lifts machine code to a higher level of abstraction (i.e., ACL2 functions), given the operational semantics of the ISA. Stateman manages large terms that represent machine states — it reduces the overhead of simplifying these terms by projecting out only the relevant parts.

6 Related Work

Our main thrust for developing an x86 ISA formal model is to enable program verification. Though program verification has a long history, with Turing's 1949 paper [82] being one of the earliest works, we briefly discuss only those efforts that involved the development and validation of formal simulators for processors.

Our modeling and reasoning strategy has been heavily influenced by the CLI stack [14], even though it was composed of systems that are simpler than modern ones. The CLI stack included NQTHM-based formal simulators of both the gate-level design and the ISA of a microprocessor called FM9001 [83], an assembler with linking loader for a language called Piton [84] that targets this microprocessor, and a higher-level language, micro-Gypsy [85], that targets Piton. Each of these simulators was validated and used as a reasoning framework to prove the correctness of the system "above" it. Another project that influenced our work was the formalization [86] of most of the user-mode instruction set of a commercial Motorola MC68020 microprocessor by Boyer and Yu in NQTHM; this formal model was used to verify machine code produced by compiling the Berkeley string library using GCC. Our work on formalizing the Y86 [87], a simple 32-bit x86-like processor that was developed by Bryant and O'Hallaron for pedagogical purposes, can be called the precursor of our x86 ISA model; we used our ACL2-based Y86 model as a prototype to evaluate the design decisions we made for our x86 ISA model. This Y86 model is also released [88] as a part of the ACL2 Community Books.

Many simulators have been built for semi-formal and formal purposes. Formal models of processor ISAs have been used as a target specification for microprocessor design verification [9, 10, 27, 83]. Rockwell Collins built a symbolic simulator [89] for their JEM1 microprocessor in the PVS [90] theorem proving system to detect microcode errors. Rockwell also published studies [91, 92] from an engineering standpoint of how to efficiently simulate formal models of processors in ACL2, again using the JEM1 microprocessor as an example. Formal models of mainstream commercial multiprocessors like ARM [93], PowerPC [94], and x86 [95–97] have been built in the HOL [98] theorem prover to develop rigorous semantics of relaxed-memory concurrency that are provided by these modern architectures. HOL-based ISA-level specifications of these mainstream processors (e.g., ARMv7 [99]) have been used to verify machine-code programs automatically using Myreen's "decompilation into logic" technique [56, 59, 100], which reduces the problem of reasoning about machine code to reasoning about simpler logic functions. The verification effort for seL4 [101, 102], the "world's first operating-system kernel with an end-to-end proof of implementation correctness and security enforcement", was initially done on formal models of ARMv6 and x86 using Isabelle/HOL. Morrisett et al. developed a framework for software fault isolation [55] that involved building a Coq-based x86

ISA specification that can be used for machine-code verification. Morrisett has also been focused on building scalable formal models for reasoning [103]. The Coq proof assistant [104] was used by Feng et al. to build a simplified formal model of the x86 processor in order to verify machine code using domain-specific and separation logics [105]. Shao's recent efforts to develop and certify clean-slate OS kernels [106] has also involved modeling processor architectures in Coq. There have been investigations into developing domain-specific languages [107, 108] to facilitate clear and precise specification of ISAs, even by non-experts, to reduce the possibility of introducing modeling errors.

Simulators used for testing have also received attention — so much so that formal methods have been employed to ensure that they are faithful to the hardware. A Coq formalization [109] of the ARM instruction set and addressing modes was done as a part of a project to certify a System-on-Chip simulator `SimSoC`. A C model of `SimSoC` was obtained in Coq using the C semantics provided by the CompCert project [110]. Parts of these two models were then proved equivalent in Coq.

7 Conclusion

We have presented our executable model of the x86 ISA, which formalizes an interpreter-style operational semantics in ACL2. The contribution of our work is a unified x86 ISA model for both simulation and formal analysis of x86 machine-code programs. It can also be used as a target specification to verify whether a micro-architecture implements this ISA. Our unified model offers two modes of operation, which allow choosing the level of rigor with which analysis will be performed. Formal models written in most higher-order theorem proving systems require the extraction of executable specifications for validation via co-simulations; thus, trusting or verifying the extraction process is necessary. Our ACL2-based formal specifications of the x86 ISA are directly executable; we also provide tools that aid in dynamic program instrumentation. Though higher-order systems offer more expressibility, applications like specifying processor ISAs are often more "natural" to do in first-order logic. Models written in first-order logic look very like the procedural code developers without experience in formal methods work with, which facilitates the adoption of tools based on these models.

Properties about our model are certified using ACL2. We have proved many properties (e.g., full functional correctness, memory usage analysis) of several application-level programs. We are currently working on making our framework more amenable to reasoning about programs with supervisor privileges — such programs directly impact system security. At the same time, we continue to extend our model by supporting more features of the x86 ISA. The latest version of our framework is available online [12].

Our work is in the spirit of the CLI and ProCoS [111, 112] projects. In this chapter, instead of verification techniques, we have concentrated on the engineering aspect of formal tool construction. Though developing and improving verification techniques is critical to achieve software reliability, it is equally important to examine our tools. Modern systems are often large and complex, as a result of which there is a considerable amount of engineering involved before the groundwork for useful research can be laid. We hope that documenting the insights we gained while building our formal framework will encourage others to discuss and share their own engineering efforts.

Acknowledgements This project was supported by the DARPA CRASH program under N66001-10-2-4087, and preparation of this chapter was also partially supported by the DARPA MUSE program under FA8750-15-C-0007. We are currently supported by NSF SaTC program under CNS-1525472. We thank Marijn Heule for his feedback on the chapter.

References

1. von Neumann, J.: First draft of a report on the EDVAC. IEEE Ann. Hist. Comput. **15**(4), 27–75 (1993). http://doi.ieeecomputersociety.org/10.1109/85.238389
2. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. J. Math. **58**(345–363), 5 (1936)
3. Rojas, R.: Konrad Zuse's legacy: the architecture of the Z1 and Z3. Ann. Hist. Comput. IEEE **19**(2), 5–16 (1997)
4. Intel Manuals (September, 2015) Intel 64 and IA-32 Architectures Software Developer's Manuals. Order Number: 325462-056US
5. Intel (Accessed: October, 2015.) Intel Developer Zone - ISA Extensions. See https://software.intel.com/en-us/isa-extensions/
6. Kaufmann, M., Moore, J.S.: (Accessed: 2015) ACL2 home page. (see http://www.cs.utexas.edu/users/moore/acl2)
7. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston (2000)
8. Boyer, R.S., Kaufmann, M., Moore, J.S.: The boyer-moore theorem prover and its interactive enhancement. Comput. Math. Appl. **29**(2), 27–62 (1995). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.7689
9. Sawada, J., Hunt Jr., J.W.: Verification of FM9801: an out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. Form. Methods Syst. Des. **20**(2), 187–222 (2002). http://dl.acm.org/citation.cfm?id=584665
10. Hunt Jr., W.A.: FM8501: A Verified Microprocessor, LNAI, vol. 795. Springer (1994)
11. Section 2.2.10: Intel 64 Architecture, Vol. 1, Intel 64 and IA-32 Architectures Software Developer's Manual. (September, 2015) Order Number: 325462-056US
12. x86isa Books in the ACL2 Community Books Project on Github (Accessed: October, 2015). See https://github.com/acl2/acl2/tree/master/books/projects/x86isa
13. x86isa ACL2 Books (Accessed: October, 2015) Documentation of the bleeding-edge ACL2-based model of the x86 ISA. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html?topic=ACL2____X86ISA
14. Bevier, W.R., Hunt Jr., W.A., Moore, J.S., Young, W.D.: Special issue on system verification. J. Autom. Reason. **5**(4), 409–530 (1989)
15. Kaufmann, M., Moore, J.S., Ray, S., Reeber, E.: Integrating external deduction tools with ACL2. J. Appl. Log. **7**(1), 3–25 (2009). doi:10.1016/j.jal.2007.07.002, http://www.sciencedirect.com/science/article/pii/S1570868307000602, special Issue: Empirically Successful Computerized Reasoning
16. Swords, S.: A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover. PhD thesis, Department of Computer Sciences, The University of Texas at Austin (2010). http://repositories.lib.utexas.edu/handle/2152/ETD-UT-2010-12-2210

17. Glucose SAT Solver (Accessed: October, 2015). http://www.labri.fr/perso/lsimon/glucose/
18. Minisat SAT Solver (Accessed: October, 2015). http://minisat.se/
19. ACL2 System and Books Repository on Github (Accessed: October, 2015). See https://github.com/acl2/acl2
20. Davis, J., Kaufmann, M.: Industrial-Strength Documentation for ACL2. EPTCS 152:9–25 (2014). http://www.cs.utexas.edu/users/kaufmann/talks/acl2-workshop-2014/acl2-14-davis-kaufmann.pdf
21. Kaufmann, M., Moore, J.S.: (Accessed: October, 2015) ACL2 documentation. See http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____ACL2
22. 2005 ACM Software System Award (2005) The Boyer-Moore Theorem Prover. (see http://awards.acm.org/software_system/)
23. ACL2 Applications (Accessed: October, 2015). See http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____INTERESTING-APPLICATIONS
24. Centaur Technology (Accessed: October 2015). http://www.centtech.com
25. FV Group at Centaur (Accessed: October, 2015). http://fv.centtech.com
26. Davis, J., Slobodova, A., Swords, S.: Microcode verification — another piece of the microprocessor verification puzzle. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving, Lecture Notes in Computer Science, vol. 8558, pp. 1–16. Springer International Publishing (2014). doi:10.1007/978-3-319-08970-6_1
27. Hunt Jr., W.A., Swords, S., Davis, J., Slobodova, A.: Use of formal verification at centaur technology. In: Hardin, D.S. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 65–88. Springer (2010). https://www.cs.utexas.edu/~jared/publications/2010-hardin-centaur.pdf
28. Flatau, A., Kaufmann, M., Reed, D.F., Russinoff, D., Smith, E.W., Sumners, R.: Formal verification of microprocessors at AMD. In: 4th International Workshop on Designing Correct Circuits (DCC 2002), Grenoble, France (2002)
29. Russinoff, D.M.: A case study in formal verification of register-transfer logic with ACL2: the floating point adder of the AMD athlon TM processor. In: Formal Methods in Computer-Aided Design, pp. 22–55. Springer (2000)
30. Russinoff, David: Computation and formal verification of SRT quotient and square root digit selection tables. IEEE Trans. Comput. **62**(5), 900–913 (2013)
31. Russinoff, D., Kaufmann, M., Smith, E., Sumners, R.: Formal verification of floating-point RTL at AMD using the ACL2 theorem prover. In: Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, Paris, France (2005)
32. Reeber, E., Sawada, J.: Combining ACL2 and an automated verification tool to verify a multiplier. In: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications, pp. 63–70. ACM (2006)
33. Sawada, J., Reeber, E.: ACL2SIX: a hint used to integrate a theorem prover and an automated verification tool. In: Formal Methods in Computer Aided Design, 2006. FMCAD '06, pp. 161–170 (2006). doi:10.1109/FMCAD.2006.3
34. Sawada, J., Gamboa, R.: Mechanical verification of a square root algorithm using Taylor's theorem. In: Formal Methods in Computer-Aided Design, pp. 274–291. Springer (2002)
35. Sawada, J., Sandon, P., Paruthi, V., Baumgartner, J., Case, M., Mony, H.: Hybrid verification of a hardware modular reduction engine. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD Inc, Austin, TX, FMCAD '11, pp. 207–214 (2011). http://dl.acm.org.ezproxy.lib.utexas.edu/citation.cfm?id=2157654.2157686
36. O'Leary, J.W., Russinoff, D.M.: Modeling algorithms in SystemC and ACL2. In: Verbeek, F., Schmaltz, J. (eds.) Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol. 152, pp. 145–162 (2014). doi:10.4204/EPTCS.152.12
37. Coglio, A.: Second-order functions and theorems in ACL2. In: Kaufmann, M., Rager, D.L. (eds.) Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its

Applications, Austin, Texas, USA, 1-2 October 2015, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol. 192, pp. 17–33 (2015). doi:10.4204/EPTCS.192.3

38. Selfridge, S., Smith, E.W.: Polymorphic types in ACL2. In: Verbeek, F., Schmaltz, J., (eds.) Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol. 152, pp. 49–59 (2014). doi:10.4204/EPTCS.152.4
39. Rager, D.L., Ebergen, J., Lee, A., Nadezhin, D., Selfridge, B., Chau, C.K.: A brief introduction to oracle's use of ACL2 in verifying floating-point and integer arithmetic. In: Kaufmann, M., Rager, D.L. (eds.) Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol. 192 (2015)
40. Greve, D.: Address enumeration and reasoning over linear address spaces. In: Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004) (2004)
41. Greve, D., Richards, R., Wilding, M.: A summary of intrinsic partitioning verification. In: 5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004), Austin, TX (2004)
42. Hardin, D.S., Smith, E.W., Young, W.D.: A robust machine code proof framework for highly secure applications. In: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications, pp. 11–20. ACM, New York, NY, USA, ACL2 '06 (2006). doi:10.1145/1217975.1217978
43. CLHS (Accessed: October, 2015) Common Lisp HyperSpec. http://www.lispworks.com/reference/HyperSpec/index.html
44. ACL2 Feature: Guards (Accessed: October, 2015). See http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____GUARD
45. Greve, D.A., Kaufmann, M., Manolios, P., Moore, J.S., Ray, S., Ruiz-Reina, J.-L., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. J. Funct. Program. **18**(1) (2008)
46. System Class Integer (Accessed: October, 2015) CLHS, Common Lisp HyperSpec. http://www.lispworks.com/documentation/HyperSpec/Body/t_intege.htm
47. ACL2 Feature: Trust Tags (Accessed: October, 2015). See http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____DEFTTAG
48. ACL2 Feature: Untouchable Functions (Accessed: October, 2015). See http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____PUSH-UNTOUCHABLE
49. The CompCert Project (Accessed: October, 2015). The CompCert C Compiler. http://compcert.inria.fr/compcert-C.html
50. Linux Memory Management (Accessed: October, 2015). Linux System Administrators Guide. http://www.tldp.org/LDP/sag/html/memory-management.html
51. McKusick, M.K., Neville-Neil, G.V., Watson, R.N.M.: Chapter 6: Memory Management. The Design and Implementation of the FreeBSD Operating System, Addison Wesley Professional (2014)
52. Kerrisk, M.: The Linux Programming Interface: A Linux and UNIX System Programming Handbook, 1st edn. No Starch Press, San Francisco (2010)
53. Windows Memory Management (Accessed: October, 2015.) Windows System Administrators Guide. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366779(v=vs.85).aspx
54. Yu, D., Shao, Z.: Verification of safety properties for concurrent assembly code. In: Proceedings of 2004 International Conference on Functional Programming (ICFP'04) (2004)
55. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B., Gan, E.: RockSalt: better, faster, stronger SFI for the x86. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 395–404. ACM, PLDI '12 (2012). doi:10.1145/2254064.2254111

56. Myreen, M.O.: Formal Verification of Machine-code Programs. PhD thesis, University of Cambridge, Computer Laboratory, Trinity College (2008). http://www.cl.cam.ac.uk/~mom22/thesis.pdf
57. Smith, E.W.: Axe, An Automated Formal Equivalence Checking Tool For Programs. PhD thesis, Department of Computer Science, Stanford University (2011)
58. Moore, J.S.: (Accessed: October, 2015) Codewalker. See https://github.com/acl2/acl2/books/projects/codewalker
59. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic - improved. In: Formal Methods in Computer-Aided Design (FMCAD), 2012, pp. 78–81 (2012). http://www.cs.utexas.edu/~hunt/FMCAD/FMCAD12/016.pdf
60. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)
61. Lawton, K.P.: Bochs: A Portable PC Emulator for Unix/X. Linux J 1996(29es) (1996). http://dl.acm.org/citation.cfm?id=326350.326357
62. Unicorn (Accessed: October, 2015) Unicorn: Slides from BlackHat USA 2015. http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf
63. AMD Manuals (Accessed: October, 2015.) AMD64 Architecture: Developer Guides, Manuals and ISA Documents. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/
64. Goel, S., Hunt Jr., W.A., Kaufmann, M.: Abstract stobjs and their application to ISA modeling. In: Proceedings of the ACL2 Workshop 2013, EPTCS 114, pp. 54–69 (2013). http://arxiv.org/pdf/1304.7858.pdf
65. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. SIGPLAN Not **40**(6), 190–200 (2005). doi:10.1145/1064978.1065034
66. Kaufmann, M., Hunt Jr., W.A.: Towards a Formal Model of the x86 ISA. Technical report, Department of Computer Science, University of Texas at Austin, Technical Report TR-12-07 (May 2012). see http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2075.pdf
67. Moore, J.S.: (Accessed: October, 2015.) Mechanized Operational Semantics. Lectures in the Marktoberdorf Summer School (August 5-16, 2008). See http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html
68. Boyer, R.S., Moore, J.S.: Single-threaded Objects in ACL2. In: Krishnamurthy, S., Ramakrishnan, C.R. (eds.) Practical Aspects of Declarative Languages (PADL), vol. 2257, pp. 9–27. Springer, LNCS (2002)
69. Hunt Jr., W.A., Kaufmann, M.: A formal model of a large memory that supports efficient execution. In: Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2012, Cambrige, UK, October 22–25) (2012). http://www.cs.utexas.edu/~hunt/FMCAD/FMCAD12/014.pdf
70. ACL2 Feature: Abstract Stobjs (Accessed: October, 2015). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html?topic=ACL2____DEFABSSTOBJ
71. Kaufmann, M., Sumners, R.: Efficient rewriting of operations on finite structures in ACL2. In: ETAPS 2002: European joint conference on theory and practice of software. Satellite workshop, pp. 141–150 (2002)
72. Greve, D.: Scalable normalization for heap manipulating functions. In: ACL2 Workshop 2007 (2007). http://www.cs.uwyo.edu/~ruben/acl2-07/uploads/Main/017.pdf
73. Goel, S., Hunt Jr., W.A., Kaufmann, M., Ghosh, S.: Simulation and formal verification of x86 machine-code programs that make system calls. In: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD'14), pp. 18:91–98 (2014). http://dl.acm.org/citation.cfm?id=2682923.2682944
74. Bitops (Accessed: October, 2015) An ACL2 Library for Reasoning about Bit-Vector Arithmetic. See http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____BITOPS
75. Rager D.L.: (Accessed: October, 2015.) Maintaining community [in] sanity with Jenkins and Github. ACL2 Workshop 2015, Rump Session Talk. See https://www.cs.utexas.

edu/users/moore/acl2/workshop-2015/rump-session-abstracts.html#rager and http://leeroy.defthm.com/

76. Matz, M., Hubicka, J., Jaeger, A., Mitchell, M.: Chapter 4: Object Files in System V Application Binary Interface. AMD64 Architecture Processor Supplement, Draft v0 99 (2005)
77. Mach-O File Format (Accessed: October, 2015.) OS X ABI Mach-O File Format Reference. Mac Developer Library. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html
78. Swords, S., Davis, J.: Bit-blasting ACL2 theorems. In: Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3–4, 2011, pp. 84–102 (2011). doi:10.4204/EPTCS.70.7
79. Anderson, S.: (Accessed: 2015) Bit Twiddling Hacks. See http://graphics.stanford.edu/~seander/bithacks.html
80. Goel, S., Hunt Jr., W.A.: Automated code proofs on a formal model of the x86. In: Verified Software: Theories, Tools, Experiments (VSTTE'13). Lecture Notes in Computer Science, vol. 8164, pp. 222–241. Springer, Berlin, Heidelberg (2014). doi:10.1007/978-3-642-54108-7_12
81. Moore, J.S.: Stateman: using metafunctions to manage large terms representing machine states. In: Kaufmann, M., Rager, D.L. (eds.) Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol. 192, pp. 93–109 (2015). doi:10.4204/EPTCS.192.8
82. Turing, A.M.: Checking a Large Routine, pp. 67–69 (1949). http://www.turingarchive.org/browse.php/B/8
83. Hunt Jr., W.A.: Microprocessor design verification. J. Autom. Reason. **5**(4), 429–460 (1989). http://www.cs.utexas.edu/~boyer/ftp/cli-reports/048.pdf
84. Moore, J.S.: Piton: A Mechanically Verified Assembly-Level Language. Automated Reasoning Series. Kluwer Academic Publishers (1996)
85. Young, William D.: A mechanically verified code generator. J. Autom. Reason. **5**(4), 493–518 (1989)
86. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. J. ACM **43**(1), 166–192 (1996). http://dl.acm.org/citation.cfm?id=227603
87. Bryant, R.E., O'Hallaron, D.R.: Chapter 4: Processor Architecture, of Computer Systems: A Programmer's Perspective. Prentice-Hall (2003)
88. Goel, S., Hunt Jr., W.A., Kaufmann, M., Krug, R.: (Accessed: 2015) y86 Specifications in the ACL2 Community Books. See https://github.com/acl2/acl2/tree/master/books/models/y86
89. Greve, D.A.: Symbolic simulation of the JEM1 microprocessor. In: Gopalakrishnan, G., Windley, P., (eds.) Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 1522, pp. 321–333. Springer, Berlin, Heidelberg (1998). doi:10.1007/3-540-49519-3_21
90. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer, Saratoga, NY (1992)
91. Greve, D., Wilding, M., Hardin, D.: High-speed, analyzable simulators. In: Computer-Aided Reasoning, pp. 113–135. Springer (2000)
92. Wilding, M., Greve, D., Hardin, D.: Efficient simulation of formal processor models. Form. Methods Syst. Des. **18**(3), 233–248 (2001)
93. Fox, A.: Formal specification and verification of ARM6. In: Theorem Proving in Higher Order Logics, pp. 25–40. Springer (2003)
94. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and ARM multiprocessor machine code. In: Proceedings of DAMP 2009: the 4th Workshop on Declarative Aspects of Multicore Programming, ACM, New York, NY, USA, 553091 (2009)
95. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010) (Research Highlights)

96. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674, pp. 391–407 (2009)
97. Sarkar, S., Sewell, P., Zappa Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M., Alglave, J.: The semantics of x86-CC multiprocessor machine code. In: Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 379–391 (2009). doi:10.1145/1594834.1480929
98. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993)
99. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C., (eds.) Interactive Theorem Proving, Lecture Notes in Computer Science, vol. 6172, pp. 243–258. Springer, Berlin (2010). doi:10.1007/978-3-642-14052-5_18
100. Myreen, M.O., Gordon, M., Slind, K.: Machine-code verification for multiple architectures - an application of decompilation into logic. In: Formal Methods in Computer-Aided Design, 2008. FMCAD '08, pp. 1–8 (2008). doi:10.1109/FMCAD.2008.ECP.24, http://www.cl.cam.ac.uk/~mom22/decomp.pdf
101. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220. ACM (2009). http://www.sigops.org/sosp/sosp09/papers/klein-sosp09.pdf
102. sel4: General Dynamics C4 Systems (Accessed: October, 2015). http://sel4.systems/
103. Morrisett, G.: Scalable formal machine models. In: Proceedings of the Second International Conference on Certified Programs and Proofs, pp. 1–3. Springer, Berlin, Heidelberg, CPP'12 (2012). doi10.1007/978-3-642-35308-6_1
104. Coq Proof Assistant (Accessed: October, 2015). http://coq.inria.fr/
105. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. J. Autom. Reason. **42**(2), 301–347 (2009). http://flint.cs.yale.edu/flint/publications/aimjar.pdf
106. Shao, Z.: Clean-slate development of certified OS kernels. In: Proceedings of the 2015 Workshop on Certified Programs and Proofs, pp. 95–96. ACM, New York, NY, USA, CPP '15 (2015). doi:10.1145/2676724.2693180
107. Fox, A.: Directions in ISA Specification. Interactive Theorem Proving (ITP), pp. 338–344 (2012). https://www.cl.cam.ac.uk/~acjf3/papers/itp12.pdf
108. Degenbaev, U.: Formal Specification of the x86 Instruction Set Architecture. PhD thesis, Universität des Saarlandes (2012). http://rg-master.cs.uni-sb.de/publikationen/UD11.pdf
109. Shi, X.: Certification of an instruction set simulator. PhD thesis, Université de Grenoble (2013)
110. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf
111. Bjørner, D.: A ProCoS project description. International Conference on AI and Robotics, North Holland (1989)
112. Bowen, J.P.: A ProCoS II project description: ESPRIT Basic Research project 7071. Bull. Eur. Assoc. Theor. Comput. Sci. (EATCS) **50**, 128–137 (1993)

Advances in Connection-Based Automated Theorem Proving

Jens Otten and Wolfgang Bibel

Abstract Automatic reasoning tools play an important role when developing provably correct software. Both main approaches, program verification and program synthesis employ automated reasoning tools, more specifically, automated theorem provers. Besides classical logic, non-classical logics are particularly relevant in this field. This chapter presents calculi to automate theorem proving in classical and some important non-classical logics, namely first-order intuitionistic and first-order modal logics. These calculi are based on the connection method, which permits a goal-oriented and, hence, a more efficient proof search. The connection calculi for these non-classical logics extend the calculus for classical logic in an elegant and uniform way by adding so-called prefixes to atomic formulae. The leanCoP theorem prover is a very compact PROLOG implementation of the connection calculus for classical logics. We present details of the implementation and describe some basic techniques to improve its efficiency. leanCoP is adapted to non-classical logics by integrating a prefix unification algorithm, which depends on the specific logic. This results in leading theorem provers for the aforementioned non-classical logics.

1 Introduction

Information Technology (IT) has been penetrating literally all areas of our society. The essential building blocks of IT are algorithms coded in hardware or software. The tools for hardware design as well as for software production have become impressively powerful indeed. Their outcomes are engineering constructs of an

J. Otten
University of Oslo, Oslo, Norway
e-mail: jeotten@ifi.uio.no

J. Otten
University of Potsdam, Potsdam, Germany

W. Bibel (✉)
Darmstadt University of Technology, Darmstadt, Germany
e-mail: bibel@gmx.net

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_9

unprecedented complexity. The correctness of these systems to a certain degree is guaranteed by ingenious and automated test methods.

While we all use systems of this kind on a daily basis, we tend to ignore their actual degree of complexity, for which reason we want to remind ourselves at this point that, for instance, actual operating systems or computing platforms today comprise hundreds of millions of lines of code (loc). The applications running on top of these platforms add to this order of magnitude in (extensional) complexity even further. How far can we trust systems this large and complex?

Unfortunately, experience shows in fact that any of these systems is full of (semantic and syntactic) bugs. Occasionally, these bugs have consequences which are embarrassing at the very least, occasionally extremely costly and sometimes even the cause for injury or death to people. Reference [1] lists five most embarrassing software bugs including the well-known Pentium FDIV bug and the disintegration of the $655-million Mars Climate Orbiter (in 1998). Further disasters caused by bugs were the crash of Ariane 5 (1996) and of an Airbus A400M Atlas cargo plane on a test flight with four people killed (2015, see [2]). Apparently, there is no guarantee for preventing a future disaster due to a bug killing many more people and causing further huge damages.

There is a second major aspect to this downside of current IT. Despite an enormous methodological improvement of the processes producing hardware and software, software projects that are large, complicated, poorly specified, or involve unfamiliar aspects, are still vulnerable to large, unanticipated problems, often leading to spectacular and costly failures (e.g. [3]). Generally, the software projects failure rate is much too high still and extensive delays are commonplace, with huge and costly consequences for the industry.

Has theory a remedy in store for these two deplorable aspects of IT? In principle, it has. In fact there are two major perspective routes for ending up with more reliable systems to begin with, known as verification and synthesis. The idea behind verification is to let a verifier check the correctness of a system against its specification. While this is possible in theory, it has remained an illusion to expect larger software projects to produce as a by-product a complete system specification, needed by the verifier. Just imagine the challenge to fully specify an operating system with hundreds of millions of loc in order to understand why, in practice and for large systems, this will remain an illusion. Smaller systems, however, have been fully and successfully verified already [26, 30, 36, 43].

Program synthesis follows the more direct route towards producing correct software. It starts from the project requirements presented in some informal way which are assumed to be transformed somehow, possibly aided with system support, into a precise specification in some formal language. The resulting formal code then in turn is automatically synthesized into an efficiently executable code which is correct under the proviso that the specification as well as the synthesizer both are correct. The problem with this approach in general — and apart from the difficulties involved in the transformation just described — is the extreme intellectual challenge involved in automating (and verifying) the synthesis step. To a limited extent though, synthesis has already been quite successful. Namely, the techniques used in hardware production are to a significant extent exactly of this nature. Similarly, popular techniques in software production such as model-driven engineering (or model-driven software development), the use of domain specific languages or modelling languages (such as UML), and so forth do already feature automatic synthesis aspects to some limited extent. Also, logical programming languages such as PROLOG have substantially narrowed the gap between a formal system specification and its executable code written in such a language, laying part of the synthesis burden on the program interpreter or compiler. However, while all these approaches may be seen as major steps towards more reliable systems, the route towards the ultimate synthesis vision has remained a truly challenging one.

To sum up so far, given the utmost importance of IT and at the same time the severe problems with IT systems, we are still faced with the challenge to find a way out of this urging dilemma since neither the practical solutions found so far nor the two theoretically possible routes have brought a satisfactory solution as yet. Hence the old vision of building Provably Correct Systems (ProCoS) has by far not lost its attractions and has remained extremely relevant.

A crucial component in both verification and synthesis is a theorem prover for some logic [12, 20, 24, 58]. This is why Automated Deduction (AD) or Automated Theorem Proving (ATP) lies at the heart of the ProCoS vision. The field of ATP has in fact made remarkable progresses in the last decades and the resulting systems are in use in numerous applications, including verification and synthesis. Yet, the problem underlying ATP seems so hard that we will have to go still a long way to reach the next higher level of performance (see Sect. 6 in this chapter as well as [16]). This is true even for a popular logic such as classical first-order logic (fol), let alone more involved logics. In the context of programming, logics other than fol are deemed necessary though. In fact, in the formal specification of an IT system, which typically is dynamic by nature, fol seems to be rather inconvenient for this purpose since it allows to represent transitions in time in an indirect way only. Non-classical or higher-order logics along with corresponding theorem provers are deemed more convenient for this purpose. Unfortunately, ATP in non-classical logics has not received the same level of attention as the one in classical fol. In part, the contributions reported in the present chapter are an attempt to make up for this neglect.

Concretely, we present a number of theorem provers for various logics some of which are outperforming any of its competitors internationally. They all belong to the family of provers uniformly designed on the basis of the leanCoP (lean Connection Prover) technology, originally developed for classical fol and following a separate and unique line of research in ATP (see Sect. 5). In fact, since theorem provers are themselves software systems which should be correct as well according to our ProCoS vision, these provers are based on a mathematically precise formalism serving as their specification. From this formal specification it is only a rather small step to the actual code written in PROLOG. The correctness proof for this step is rather straightforward in each case [51]. In other words, our provers themselves are provably correct systems as desired.

True, our provers are small systems indeed (comprising a few dozens of loc only), hence the term “lean”. They are so by intention. Competitive provers with similar performance in comparison sometimes feature hundreds of thousands loc, serving exactly the same purpose. In other words, the comparable intensional complexity of an algorithm can be represented in a vast variety of different extensional complexities. The lean extensional complexity version is accessible to formal correctness proofs while the huge one is not. This is one of the reasons why we opt for the lean approach. It is thereby understood that an optimization of the PROLOG code into some low-level code could of course be followed once the PROLOG code has settled to a stable one, whereby the optimizer should consist of verified code as well, of course. These explanations of our approach demonstrate that, apart from presenting tools in this chapter relevant for the ProCoS vision, these themselves at the same time may be seen as a model for how to follow this kind of an approach towards a provably correct software of high intensive complexity, possibly extended to high extensive complexity thereafter.

The chapter is organized as follows. Section 2 introduces basic concepts and the matrix characterization of logical validity. Section 3 presents the clausal and the non-clausal connection calculus for classical logic and some basic optimization techniques. In Sect. 4 connection calculi for first-order intuitionistic and first-order modal logics are introduced. Section 5 describes compact PROLOG implementations that are based on the presented connection calculi. In Sect. 6 we give a brief history of the line of research in ATP to which this chapter contributes. In addition we outline there some of the steps which are expected to be taken along this line in the future. Section 7 concludes with a summary and a brief outlook on further research.

2 Preliminaries

This section provides a brief overview of classical and non-classical logics, and presents the matrix characterization of logical validity, which is the basis for the connection calculi presented in Sects. 3 and 4.

2.1 Classical Logic

The reader is assumed to be familiar with the language of classical first-order logic, see, e.g., [8, 54, 62]. In this chapter the letters P is used to denote predicate symbols, f to denote function symbols and x, X to denote variables. Terms are denoted by t and are built from functions, constants and variables.

An *atomic formula*, denoted by A, is built from predicate symbols and terms. The connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ denote negation, conjunction, disjunction and implication, respectively. A *(first-order) formula*, denoted by F, G, H, consists of atomic formulae, the connectives and the existential and universal quantifiers, denoted by $\forall$ and $\exists$, respectively. A *literal* L has the form A or $\neg A$. The *complement* $\overline{L}$ of a literal L is A if L is of the form $\neg A$, and $\neg L$ otherwise. A formula in *clausal form* has the form $\exists x_1 \ldots \exists x_n (C_1 \vee \ldots \vee C_n)$, where each C_i is a clause. For classical logic, every formula F can be translated into an equivalent formula F' in clausal form.

2.2 Non-Classical Logics

Intuitionistic logic [23] and *modal logics* [9] are popular *non-classical* logics. Intuitionistic and classical logic share the same *syntax*, i.e. formulae in both logics use the same connectives and quantifiers, but their *semantics* is different. For example, the formula

$$man(Socrates) \vee \neg man(Socrates) \tag{1}$$

is valid in classical logic, but not in intuitionistic logic. This property holds for all formulae of the form $P \vee \neg P$ for any proposition P. In classical logic this formula is valid as P or $\neg P$ is true whether P is true or not true. The semantics of intuitionistic logic requires a proof for P or for $\neg P$. As this property neither holds for P nor for $\neg P$, the formula is not valid in intuitionistic logic. For this reason intuitionistic logic is also called *constructive logic*. Every formula that is valid in intuitionistic logic is also valid in classical logic, but not vice versa.

Modal logics extend the language of classical logic by the *modal operators* $\Box$ and $\Diamond$ representing *necessarily* and *possibly*, respectively. For example, the proposition "if Plato is necessarily a man, then Plato is possibly a man" can be represented by the *modal formula*

$$\Box\, man(Plato) \Rightarrow \Diamond\, man(Plato)\,. \tag{2}$$

The *Kripke semantics* of the (standard) modal logics is defined by a set of *worlds* and a binary *accessibility relation* between these worlds. In each single world the classical semantics applies to the classical connectives, whereas the modal operators $\Box$ and $\Diamond$ are interpreted with respect to accessible worlds. There is a broad range of different modal logics and the properties of the accessibility relation specify the particular modal logic. Thus, the validity of a formula depends on the chosen modal logic. For example, the modal formula 2 is valid in all (standard) modal logics.

2.3 Matrix Characterisation

The general questioning in ATP is to provide an answer to the question as to whether a given formula F' is a logical consequence of a given set of formulae $\{F_1, F_2, \ldots, F_n\}$. According to the *deduction theorem*, this problem can be reduced to the problem of determining whether the formula $F_1 \wedge F_2 \wedge \ldots \wedge F_n \Rightarrow F'$ is valid.

The matrix characterization of logical validity considers the formula to be in a certain form, often clausal form. More formally, a *matrix* of a formula consists of its clauses $\{C_1, \ldots, C_n\}$, in which each *clause* is a set of literals $\{L_1, \ldots, L_m\}$. The notion of *multiplicity* is used to encode the number of clause copies used in a connection proof. It is a function $\mu : M \to \mathbb{N}$ that assigns each clause in M a natural number specifying how many copies of this clause are considered in a proof. In the *copy of a clause* C all variables in C are replaced by new variables. M^μ is the matrix that includes these clause copies. Clause copies correspond to applications of the contraction rule in the sequent calculus [29]. In the *graphical representation* of a matrix, its clauses are arranged horizontally, while the literals of each clause are arranged vertically. The *polarity* 0 or 1 is used to represent negation in a matrix, i.e. literals of the form A and $\neg A$ are represented by A^0 and A^1, respectively,

Then, a *connection* is a set $\{A^0, A^1\}$ of literals with the same predicate symbol but different polarities. A *term substitution* σ assigns terms to variables that occur in the literals of a given formula. A connection $\{L_1, L_2\}$ with $\sigma(L_1) = \sigma(\overline{L_2})$ is called σ*-complementary*. It corresponds to a closed branch in the tableau calculus [31] or an axiom in the sequent calculus [29].

For example, the formula

$$(\,\mathit{man(Plato)} \wedge \forall X(\,\mathit{man(X)} \Rightarrow \mathit{mortal(X)}\,)\,) \Rightarrow \mathit{mortal(Plato)} \tag{3}$$

has the equivalent clausal form

$$\exists X(\,\neg \mathit{man(Plato)} \vee (\,\mathit{man(X)} \wedge \neg \mathit{mortal(X)}\,) \vee \mathit{mortal(Plato)}\,) \tag{4}$$

and its matrix is

$$M' = \{\{\mathit{man(Plato)}^1\}, \{\mathit{man(X)}^0, \mathit{mortal(X)}^1\}, \{\mathit{mortal(Plato)}^0\}\} \tag{5}$$

which has the graphical representation

$$\left[\left[\,\mathit{man(Plato)}^1\,\right] \begin{bmatrix} \mathit{man(X)}^0 \\ \mathit{mortal(X)}^1 \end{bmatrix} \left[\,\mathit{mortal(Plato)}^0\,\right] \right].$$

A *path* through a matrix $M = \{C_1, \ldots, C_n\}$ is a set of literals that contains one literal from each clause $C_i \in M$, i.e. a set $\cup_{i=1}^{n} \{L'_i\}$ with $L'_i \in C_i$. Then, the *matrix characterization* [15] states that a formula F is (classically) valid iff (if and only if) there exists (1) a multiplicity μ, (2) a term substitution σ, and (3) a set of connections S, such that every path through its matrix M^μ (attached with the multiplicity μ) contains a σ-complementary connection $\{L_1, L_2\} \in S$.

For example, in order to make $\{\mathit{man(X)}^0, \mathit{man(Plato)}^1\}$ a σ-complementary connection, the variable X needs to be substituted by *Plato*, i.e. $\sigma(X) = \mathit{Plato}$. Then every path through the matrix 5 (with multiplicity $\mu(C_i) = 1$) contains a σ-complementary connection and, hence, formula 3 is (classically) valid.

All these notions can be generalized to the *non-clausal* form case where the clauses of matrices are not just sets of literals, but may rather contain general matrices as elements as well (see Sect. 3.3). For the following characterization we assume that the term matrix characterization refers to this general case.

Any proof method that is based on the matrix characterization and operates in a connection-oriented way is called a *connection method*. The specific calculus of a connection method is called a *connection calculus*. In other words, the connection method denotes a general approach comprising many different connection calculi. This general terminology is similar to resolution. We talk of a resolution method, or simply of resolution, whenever the proof rule of resolution is involved somehow. Also in this case resolution denotes a general approach comprising many different specific resolution calculi (like, for instance, linear resolution).

3 Connection Calculi for Classical Logic

Connection calculi are a well-known basis to automate formal reasoning in classical first-order logic. Among these are the calculi introduced in [13–15], the connection tableau calculus [38], and the model elimination calculus [40]. Proof search in the connection calculus is guided by connections $\{A^0, A^1\}$, hence, it is more goal-oriented compared to the proof search in sequent or tableau calculi.

First, this section introduces a formal *clausal connection calculus* for classical logic. Afterwards the technique of *restricted backtracking* is introduced that reduces the search space in connection calculi significantly. Finally, a generalization of the connection calculus to *non-causal formulae* is presented.

3.1 The Basic Calculus

The connection calculus for classical logic to be introduced now is based on the matrix characterization of logical validity presented in Sect. 2.3. It uses a *connection-driven* search strategy in order to calculate an appropriate set of connections S. In each step of a derivation in the connection calculus a connection is identified and only paths that do not contain this connection are investigated afterwards. If every path contains a (σ-complementary) connection, the proof search succeeds and the given formula is valid. A *connection proof* can be illustrated within the graphical matrix representation. For example, the proof of matrix 5 consists of two inferences, which identify two connections:

$$\left[\left[\mathit{man(Plato)}^1 \right] \begin{bmatrix} \mathit{man(X)}^0 \\ \mathit{mortal(X)}^1 \end{bmatrix} \left[\mathit{mortal(Plato)}^0 \right] \right] .$$

$$\begin{array}{|rll|}
\hline
\textit{Axiom (A)} & \dfrac{}{\{\}, M, Path} & \\[2ex]
\textit{Start (S)} & \dfrac{C_2, M, \{\}}{\varepsilon, M, \varepsilon} & \text{and } C_2 \text{ is copy of } C_1 \in M \\[2ex]
\textit{Reduction (R)} & \dfrac{C, M, Path \cup \{L_2\}}{C \cup \{L_1\}, M, Path \cup \{L_2\}} & \text{and } \sigma(L_1) = \sigma(\overline{L_2}) \\[2ex]
\textit{Extension (E)} & \dfrac{C_2 \setminus \{L_2\}, M, Path \cup \{L_1\} \qquad C, M, Path}{C \cup \{L_1\}, M, Path} & \begin{array}{l}\text{and } C_2 \text{ is a copy of } C_1 \in M, \\ L_2 \in C_2, \text{ and } \sigma(L_1) = \sigma(\overline{L_2})\end{array} \\
\hline
\end{array}$$

Fig. 1 The clausal connection calculus for classical logic

In contrast to sequent calculi, connection calculi permit a more *goal-oriented* proof search. This leads to a significantly smaller search space and, thus, to a more efficient proof search.

A formal description of the calculus was given by Otten and Bibel [51]. The axiom and the rules of this formal *clausal connection calculus* are given in Fig. 1. The words of the calculus are tuples of the form "C, M, *Path*", where M is a matrix, C and *Path* are sets of literals or ε; C is the *subgoal clause*, *Path* is the *active path*, and σ is a *rigid* term substitution. A *clausal connection proof* of a matrix M is a clausal connection proof of $\varepsilon, M, \varepsilon$.

For example,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\}, M', \{\mathit{mortal(Plato)}^0, \mathit{man(X')}^0\}}\,A \qquad \dfrac{}{\{\}, M', \{\mathit{mortal(Plato)}^0\}}\,A}{\{\boldsymbol{man(X')}^0\}, \{\{\boldsymbol{man(Plato)}^1\}, \ldots\}, \{\mathit{mortal(Plato)}^0\}}\,E \qquad \dfrac{}{\{\}, M', \{\}}\,A}{\{\boldsymbol{mortal(Plato)}^0\}, \{\{\mathit{man(Plato)}^1\}, \{\mathit{man(X)}^0, \boldsymbol{mortal}\mathit{(X)}^1\}, \{\mathit{mortal(Plato)}^0\}\}, \{\}}\,E}{\varepsilon, \{\{\mathit{man(Plato)}^1\}, \{\mathit{man(X)}^0, \mathit{mortal(X)}^1\}, \{\mathit{mortal(Plato)}^0\}\}, \varepsilon}\,S$$

is a proof of matrix 5, termed M', in the clausal connection calculus with the term substitution $\sigma(X') = \mathit{Plato}$, in which a copy of the second clause was made. In order to prove that *Plato* as well as *Socrates* are mortal, another copy of the second clause $\{\mathit{man(X)}^0, \mathit{mortal(X)}^1\}$ would be needed, using the new variable X'' and $\sigma(X'') = \mathit{Socrates}$.

The presented clausal connection calculus is *correct* and *complete*, i.e. a formula is valid in classical logic iff there is a clausal connection proof of its matrix M [15]. The proof is based on the matrix characterization for classical logic.

Proof search in the clausal connection calculus is carried out by applying the rules of the calculus in an *analytic* way, i.e. from bottom to top, starting with $\varepsilon, M, \varepsilon$, in which M is the matrix of the given formula. At first a start clause is selected. Afterwards, connections are successively identified by applying reduction and extension rules in order to make sure that all paths through the matrix contain a σ-complementary connection. This process is guided by the active path, a subset of a path through M. During the proof search, backtracking might be required, i.e. alternative rules or rule instances have to be considered if the chosen rule or rule instance does not lead to a proof. This might happen when choosing the clause C_1 in the start and extension rules or the literal L_2 in the reduction and extension rules. The term substitution σ is calculated step by step by one of the well-known *term unification algorithms* (see, e.g. [57]) whenever a reduction or extension rule is applied.

3.2 Restricted Backtracking

In contrast to saturation-based calculi, such as resolution [57] or instance-based methods [37], standard connection calculi are not *proof confluent*, i.e. a significant amount of *backtracking* is necessary during the proof search. Backtracking is required if there is more than one rule instance applicable (see Sect. 3.1). Confluent connection calculi that have been developed so far [10, 15] have not shown an improved performance, as these calculi lose the strict goal-oriented proof search.

The idea of *restricted backtracking* is to cut off any alternative connections once a literal from the subgoal clause has been solved [46]. A literal L is called *solved* if it is the literal L_1 of a reduction or extension rule application (see Fig. 1) and in the case of the extension rule, there is also a proof for the left premise. A solved literal in the connection calculus corresponds to a closed branch in the tableau calculus.

For example, starting the proof search with the first clause of the following matrix

$$\left[\begin{bmatrix} man(X)^0 \\ mortal(X)^1 \end{bmatrix} \begin{bmatrix} man(X)^1 \\ martian(X)^1 \end{bmatrix} \left[man(Socrates)^1 \right] \left[man(Plato)^1 \right] \left[mortal(Plato)^0 \right] \right] , \quad ?$$

the first possible connection to the literal $man(X)^1$ in the second clause does not solve the literal $man(X)^0$, as the literal $martian(X)^1$ cannot be solved. But $man(X)^0$ can be solved by the second alternative connection to $man(Socrates)^1$ in the third clause, i.e.

$$\left[\begin{bmatrix} man(X)^0 \\ mortal(X)^1 \end{bmatrix} \begin{bmatrix} man(X)^1 \\ martian(X)^1 \end{bmatrix} \left[man(Socrates)^1 \right] \left[man(Plato)^1 \right] \left[mortal(Plato)^0 \right] \right] .$$

In case of backtracking, the third alternative connection to the literal $man(Plato)^1$ in the fourth clause would be considered. Restricted backtracking cuts off this third and all following alternative connections for the literal $man(X)^0$.

The potential of this approach to significantly reduce the search space becomes clear, if connection proofs for first-order formulae are analysed in a statistical way [46]. To this end the 1256 connection proofs for formulae in version 3.7.0 of the TPTP problem library [66] that are found by the automated theorem prover leanCoP are considered. It can be observed that the first connection (or rule application) that solves a literal is often the same one used in the final proof. This applies to 89% of all solved literals used within the found connection proofs. In this case, backtracking that occurs afterwards can be cut off without effecting a successful proof search, hence, it is called *non-essential* backtracking. Backtracking with alternative connections that occur before the literal is solved is still necessary and, hence, called *essential backtracking*. In the above matrix the alternative connection to the third clause is considered essential backtracking as the connection to the second clause does not solve the literal $man(X)^0$. However, the alternative connection to the fourth clause is non-essential backtracking.

Even though most literals within the connection proofs can be solved by performing only essential backtracking, a significant amount of non-essential backtracking occurs during the actual proof *search*. Restricted backtracking cuts off this non-essential backtracking [46]. As this reduces the search space significantly, the approach turns out to be very successful in practice. For example, for the formula AGT016+2 of the TPTP problem library [66], which contains more than 1000 clauses, the standard proof search requires 84 s using 312,831 inference steps. With restricted backtracking the proof search requires only 0.3 s, using 427 inference steps. Proofs are not only found faster, but many new proofs are obtained. A similar technique can also be used to restrict backtracking when selecting the *start clause* C_1 within the application of the start rule. Restricted backtracking preserves correctness of the connection calculus, as the search space is only pruned. However, completeness is lost, as can be seen by the example matrix shown above. Namely, non-essential backtracking would solve $man(X)^0$ with a connection to the fourth clause and the resulting substitution of X by *Plato* would allow to solve the second literal (by connecting it to the fifth clause) as well.

3.3 Non-clausal Calculus

Clausal connection calculi, such as the ones presented in Sect. 3.1, require the input formula in disjunctive normal (or clausal) form. Formulae that are not in clausal form have to be translated into this form. The standard transformation translates a first-order formula F into clausal form by applying the distributivity laws. In the worst case, the size of the resulting formula grows exponentially with respect to the size of the original formula F. This increases the search space significantly. Even a definitional translation [52] that introduces definitions for subformulae introduces a significant overhead for the proof search [46]. Furthermore, both clausal form translations modify the structure of the original formula F.

A *non-clausal* connection calculus [47] that works directly on the structure of the original formula does not have these disadvantages. Existing non-clausal approaches [7, 15, 32] work only on ground formulae. For first-order formulae, copies of subformulae are added iteratively, which introduces a huge redundancy into the proof search. For a more efficient proof search, clauses have to be added

Table 1 The definition of the non-clausal matrix

Type	F^{pol}	$M(F^{pol})$
atomic	A^0	$\{\{A^0\}\}$
	A^1	$\{\{A^1\}\}$
α	$(\neg G)^0$	$M(G^1)$
	$(\neg G)^1$	$M(G^0)$
	$(G \wedge H)^1$	$\{\{M(G^1)\}, \{M(H^1)\}\}$
	$(G \vee H)^0$	$\{\{M(G^0)\}, \{M(H^0)\}\}$
	$(G \Rightarrow H)^0$	$\{\{M(G^1)\}, \{M(H^0)\}\}$
β	$(G \wedge H)^0$	$\{\{M(G^0), M(H^0)\}\}$
	$(G \vee H)^1$	$\{\{M(G^1), M(H^1)\}\}$
	$(G \Rightarrow H)^1$	$\{\{M(G^0), M(H^1)\}\}$
γ	$(\forall x G)^1$	$M(G[x\backslash x^*]^1)$
	$(\exists x G)^0$	$M(G[x\backslash x^*]^0)$
δ	$(\forall x G)^0$	$M(G[x\backslash t^*]^0)$
	$(\exists x G)^1$	$M(G[x\backslash t^*]^1)$

dynamically during the proof search, similar to the approach used for copying clauses in clausal connection calculi. To this end, the clausal connection calculus is generalized and its rules are carefully extended.

The *non-clausal matrix* $M(F^{pol})$ of a formula F with polarity *pol* is a set of clauses, in which a clause is a set of literals and (sub-)matrices, and is defined inductively according to Table 1 [47]. In this table $G[x\backslash t]$ denotes the formula G in which all free occurrences of x are replaced by t. x^* is a new variable, t^* is the Skolem term $f^*(x_1, \ldots, x_n)$ in which f^* is a new function symbol and $x_1, \ldots, x_n$ are the free variables in $\forall xG$ or $\exists xG$. The *non-clausal matrix* of a formula F is the matrix $M(F^0)$. In the *graphical representation* its clauses are arranged horizontally, literals and (sub-)matrices of its clauses are arranged vertically.

For example, the formula

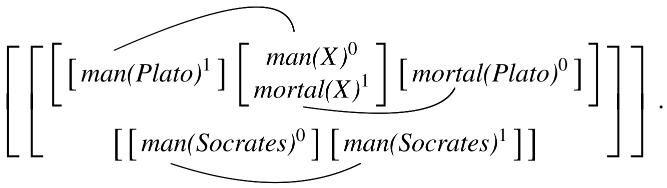
$$\begin{aligned}(\,(\,\mathit{man(Plato)} \wedge \forall X(\mathit{man(X)} \Rightarrow \mathit{mortal(X)})\,) \Rightarrow \mathit{mortal(Plato)}\,)\\ \wedge\,(\,\mathit{man(Socrates)} \vee \neg \mathit{man(Socrates)}\,)\end{aligned} \tag{6}$$

has the (simplified) non-clausal matrix

$$\begin{aligned}\{\{\{\{\mathit{man(Plato)}^1\}, \{\mathit{man(X)}^0, \mathit{mortal(X)}^1\}, \{\mathit{mortal(Plato)}^0\}\},\\ \{\{\mathit{man(Socrates)}^0\}, \{\mathit{man(Socrates)}^1\}\}\}\}\ .\end{aligned} \tag{7}$$

The definition of paths through a non-clausal matrix can be generalized in a straightforward way. All other concepts used for clausal matrices, e.g. the definitions of connections and term substitutions, remain unchanged.

For example, the non-clausal connection proof of matrix 7 using the substitution $\sigma(X) = Plato$ is illustrated in its graphical (non-clausal) matrix



The formal *non-clausal connection calculus* [47] has the same axiom, start rule, and reduction rule as the clausal connection calculus. The extension rule is restricted to so-called extension clauses and a *decomposition rule* that splits subgoal clauses into their subclauses is added. A clause C in a matrix M is an *extension clause of M with respect to* a set of literals *Path* iff

a. C contains a literal of *Path*, or
b. C is α-related to all literals of *Path* occurring in M and if C has a parent clause, it contains a literal of *Path*.

A clause C is *α-related* to a literal L iff it occurs besides L in the graphical matrix representation. For example, in the given matrix, $man(Plato)^1$ is only α-related to $man(X)^0$, $mortal(X)^1$, and $mortal(Plato)^0$. The *parent clause* of a clause C in a matrix M is a clause $C' = \{M_1, \ldots, M_n\}$ in M such that $C \in M_i$ for some $1 \le i \le n$. See [47] for the full description of the formal non-clausal connection calculus.

The non-clausal connection calculus for classical logic is *correct* and *complete*. The correctness proof is based on the non-clausal matrix characterization, completeness is proved by an embedding into the clausal connection calculus.

The *proof search* in the non-clausal connection calculus is carried out in the same way as in the clausal connection calculus. On formulae in clausal form, the non-clausal connection calculus behaves just like the clausal connection calculus. If the matrices that are used in the non-clausal connection calculus are slightly modified, the start and the reduction rule are subsumed by the decomposition and the extension rule, respectively [47]. Optimization techniques, such as positive start clauses, regularity, and restricted backtracking, can be used in the non-clausal connection calculus as well. Furthermore, the non-clausal calculus can be extended to *non-classical logics* in the same way as the clausal connection calculus (see Sect. 4).

4 Connection Calculi for Non-classical Logics

By using the notion of *prefixes* the connection calculus for classical logics can be extended to intuitionistic logic and several modal logics.

4.1 Intuitionistic Logic

Every formula F that is valid in intuitionistic logic is also valid in classical logic. The opposite direction does not hold. Hence, the three rules

$$\frac{\Gamma, G \vdash}{\Gamma \vdash \neg G, \Delta}\ \neg\textit{-right}\ , \qquad \frac{\Gamma, G \vdash H}{\Gamma \vdash G \Rightarrow H, \Delta}\ \Rightarrow\textit{-right}\ , \qquad \frac{\Gamma \vdash G[x \backslash a]}{\Gamma \vdash \forall x\, G, \Delta}\ \forall\textit{-right}$$

of the *sequent calculus for intuitionistic logic* [29] differ from the ones for classical logic. In all three rules the set of formulae Δ does not occur in the sequent of the premises anymore. During the proof search these rules are applied from bottom to top and the formulae in Δ are removed from the sequent. As these formulae might be necessary to complete the proof, the application of these rules need to be controlled. To this end, a prefix is assigned to every subformula G of a given formula F. A *prefix* is a string, i.e. a sequence of characters over an alphabet $\Phi \cup \Psi$, in which Φ is a set of *prefix variables* and Ψ is a set of *prefix constants*. Prefix constants and variables represent applications of the rules $\neg$*-right*, $\Rightarrow$*-right*, $\forall$*-right*, and $\neg$*-left*, $\Rightarrow$*-left*, $\forall$*-left*, respectively [69, 70]. Then, the prefix p of a subformula G, denoted $G\!:\!p$, specifies the sequence of these rules that have to be applied (analytically) to obtain G in the sequent. In order to preserve two atomic formulae that form an axiom in the intuitionistic sequent calculus, their prefixes need to unify. This is done by an *intuitionistic substitution* σ_J that maps elements of Φ to strings over $\Phi \cup \Psi$.

In the *matrix characterization for intuitionistic logic* it is additionally required that the prefixes of the literals in every connection unify under σ_J [70]. For a combined substitution $\sigma := (\sigma_Q, \sigma_J)$, a connection $\{L_1\!:\!p_1, L_2\!:\!p_2\}$ is σ*-complementary* iff $\sigma_Q(L_1) = \sigma_Q(\overline{L_2})$ and $\sigma_J(p_1) = \sigma_J(p_2)$. An additional *interaction condition* on σ ensures that σ_Q and σ_J are mutually consistent [70].

For intuitionistic logic there exists no equivalent clausal form for a given formula F and the original matrix characterization for intuitionistic logic does not use a clausal form. In order to adapt the existing clausal connection calculus for classical logic, Wallen's original matrix characterization has to be modified. To this end, the *skolemization* technique, originally used to eliminate *eigenvariables* in classical logic, is extended and also used for prefix constants in intuitionistic logic [44]. This allows the specification of a *clausal* matrix characterization, in which clause copies can simply be made by renaming all term and prefix variables [44]. Furthermore, there is no need for an explicit *irreflexivity test* of the *reduction ordering*. Instead, this test is realized by the *occurs check* during the term and prefix unification. For classical logic this close relationship between the reduction ordering and skolemization was first pointed out by Bibel [15]. For the extended skolemization, the same Skolem function symbol is used for instances of the same subformula, a technique that is similar to the *liberalized* δ^+*-rule* in classical tableau calculi [31].

The following description gives a formal definition of a prefixed clausal matrix for intuitionistic logic and the extended skolemization. The *prefixed matrix* $M(F^{pol}\!:\!p)$ of a *prefixed formula* $F^{pol}\!:\!p$ is a set of *prefixed clauses*, in which *pol* is a polarity and p is a prefix, and is defined inductively according to Table 2 [44]. In this table it is

Table 2 The definition of the prefixed matrix for intuitionistic logic

Type	$F^{pol} : p$	$M(F^{pol} : p)$
atomic	$A^0 : p$	$\{\{A^0 : pa^*\}\}$
	$A^1 : p$	$\{\{A^1 : pV^*\}\}$
α	$(\neg G)^0 : p$	$M(G^1 : pa^*)$
	$(\neg G)^1 : p$	$M(G^0 : pV^*)$
	$(G{\wedge}H)^1 : p$	$M(G^1 : p) \cup M(H^1 : p)$
	$(G{\vee}H)^0 : p$	$M(G^0 : p) \cup M(H^0 : p)$
	$(G{\Rightarrow}H)^0 : p$	$M(G^1 : pa^*) \cup M(H^0 : pa^*)$
β	$(G{\wedge}H)^0 : p$	$M(G^0 : p) \cup_\beta M(H^0 : p)$
	$(G{\vee}H)^1 : p$	$M(G^1 : p) \cup_\beta M(H^1 : p)$
	$(G{\Rightarrow}H)^1 : p$	$M(G^0 : pV^*) \cup_\beta M(H^1 : pV^*)$
γ	$(\forall xG)^1 : p$	$M(G[x\backslash x^*]^1 : pV^*)$
	$(\exists xG)^0 : p$	$M(G[x\backslash x^*]^0 : p)$
δ	$(\forall xG)^0 : p$	$M(G[x\backslash t^*]^0 : pa^*)$
	$(\exists xG)^1 : p$	$M(G[x\backslash t^*]^1 : p)$

$M_G \cup_\beta M_H := \{C_G \cup C_H \mid C_G \in M_G,\ C_H \in M_H\}$. x^* is a new term variable, t^* is the Skolem term $f^*(x_1, \ldots, x_n)$ in which f^* is a new function symbol and $x_1, \ldots, x_n$ are all free term and prefix variables in $(\forall xG)^0 : p$ or $(\exists xG)^1 : p$. V^* is a new prefix variable, a^* is a prefix constant of the form $f^*(x_1, \ldots, x_n)$ in which f^* is a new function symbol and $x_1, \ldots, x_n$ are all free term and prefix variables in $A^0 : p$, $(\neg G)^0 : p$, $(G{\Rightarrow}H)^0 : p$, or $(\forall xG)^0 : p$. The *intuitionistic matrix* $M(F)$ of a formula F is the prefixed matrix $M(F^0 : \varepsilon)$, in which ε is the empty string.

For example, the intuitionistic matrix of the formula

$$(\mathit{man(Plato)} \wedge \forall X(\mathit{man(X)} \Rightarrow \mathit{mortal(X)})) \Rightarrow \mathit{mortal(Plato)} \tag{8}$$

is

$$\{\{\mathit{man(Plato)}^1 : a_1 V_1\},\ \{\mathit{man(X)}^0 : a_1 V_2\, a_2(X),\ \mathit{mortal(X)}^1 : a_1 V_2 V_3\},\ \{\mathit{mortal(Plato)}^0 : a_1 a_3\}\}\ , \tag{9}$$

in which $a_1, a_2(X), a_3$ are prefix constants, and V_1, V_2, V_3 are prefix variables. Then,

$$\left[\left[\mathit{man(Plato)}^1 : a_1 V_1 \right] \begin{bmatrix} \mathit{man(X)}^0 : a_1 V_2\, a_2(X) \\ \mathit{mortal(X)}^1 : a_1 V_2 V_3 \end{bmatrix} \left[\mathit{mortal(Plato)}^0 : a_1 a_3 \right] \right]$$

is a graphical intuitionistic connection proof of matrix 9 with $\sigma_Q(X) = \mathit{Plato}$, $\sigma_J(V_1) = a_2(\mathit{Plato})$, $\sigma_J(V_2) = \varepsilon$, and $\sigma_J(V_3) = a_3$, where ε is the empty string.

The intuitionistic matrix of the formula

Axiom (A)	$\dfrac{}{\{\},M,Path}$	
Start (S)	$\dfrac{C_2,M,\{\}}{\varepsilon,M,\varepsilon}$	and C_2 is copy of $C_1{\in}M$
Reduction (R)	$\dfrac{C,M,Path{\cup}\{L_2:p_2\}}{C{\cup}\{L_1:p_1\},M,Path{\cup}\{L_2:p_2\}}$	and $\{L_1:p_1,L_2:p_2\}$ is σ-complementary
Extension (E)	$\dfrac{C_2\backslash\{L_2:p_2\},M,Path{\cup}\{L_1:p_1\} \qquad C,M,Path}{C{\cup}\{L_1:p_1\},M,Path}$	and C_2 is a copy of $C_1{\in}M$, $L_2{:}p_2{\in}C_2$, $\{L_1{:}p_1,L_2{:}p_2\}$ is σ-complementary

Fig. 2 The clausal connection calculus for intuitionistic logic

$$man(Socrates) \vee \neg man(Socrates) \tag{10}$$

is

$$\{\{man(Socrates)^0 : a_1\}, \{man(Socrates)^1 : a_2\}\} \;. \tag{11}$$

There is no substitution σ_J with $\sigma_J(a_1) = \sigma_J(a_2)$ and no connection proof of this matrix. Hence, formula 10 is *not* valid in intuitionistic logic.

The formal *clausal connection calculus for intuitionistic logic* [44] is shown in Fig. 2. It is an extension of the clausal connection calculus for classical logic, in which a prefix is added to each literal and an additional intuitionistic substitution is used to identify σ-complementary connections. An *intuitionistic connection proof* of the matrix M is a proof of ε, M, ε. The clausal connection calculus for intuitionistic logic is *correct* and *complete*, i.e. a formula F is valid in intuitionistic logic iff there is an intuitionistic connection proof of its intuitionistic matrix $M(F)$.

The intuitionistic substitution σ_J is calculated by a *prefix unification algorithm* [44]. For a given set of prefix equations $\{p_1 = q_1, \ldots, p_n = q_n\}$, an appropriate substitution σ_J is a unifier such that $\sigma_J(p_i) = \sigma_J(q_i)$ for all $1 \le i \le n$. General algorithms for string unification exist, but the following unification algorithm is more efficient, as it takes the *prefix property* of all prefixes $p_1, p_2, \ldots$ into account: for two prefixes $p_i = u_1 X w_1$ and $p_j = u_2 X w_2$ with $X \in \Phi \cup \Psi$ the property $u_1 = u_2$ holds. This reflects the fact that prefixes correspond to sequences of connectives and quantifiers within the same formula.

The prefix unification for the prefixes equation $\{p = q\}$ is carried out by applying the *rewriting rules* in Fig. 3. It is V, $\bar{V}$, $V' \in \Phi$ with $V \ne \bar{V}$, V' is a new prefix variable, $a, b \in \Psi$, $X \in \Phi \cup \Psi$, and $u, w, z \in (\Phi \cup \Psi)^*$. For rule 10 the restriction $(*)$ $u = \varepsilon$ or $w \ne \varepsilon$ or $X \in \Psi$ applies. $\sigma_J(V) = u$ is written $\{V\backslash u\}$.

The unification starts with the tuple $(\{p = \varepsilon|q\}, \{\})$. The application of a rewriting rule $E \to E', \tau$ replaces the tuple (E, σ_J) by the tuple $(E', \tau(\sigma_J))$. E and E' are prefix equations, σ_J and τ are (intuitionistic) substitutions. The unification terminates when the tuple $(\{\}, \sigma_J)$ is derived. In this case, σ_J represents a *most general unifier*. Rules can be applied non-deterministically and lead to a *minimal* set of most general

1.	$\{\varepsilon{=}\varepsilon \mid \varepsilon\}$	$\to \{\},\{\}$	6.	$\{Vu{=}\varepsilon \mid aw\}$	$\to \{u{=}\varepsilon \mid aw\},\{V\backslash\varepsilon\}$
2.	$\{\varepsilon{=}\varepsilon \mid Xu\}$	$\to \{Xu{=}\varepsilon \mid \varepsilon\},\{\}$	7.	$\{Vu{=}z \mid abw\}$	$\to \{u{=}\varepsilon \mid bw\},\{V\backslash za\}$
3.	$\{Xu{=}\varepsilon \mid Xw\}$	$\to \{u{=}\varepsilon \mid w\},\{\}$	8.	$\{Vau{=}\varepsilon \mid \bar{V}w\}$	$\to \{\bar{V}w{=}V \mid au\},\{\}$
4.	$\{au{=}\varepsilon \mid Vw\}$	$\to \{Vw{=}\varepsilon \mid au\},\{\}$	9.	$\{Vau{=}Xz \mid \bar{V}w\}$	$\to \{\bar{V}w{=}V' \mid au\},\{V\backslash XzV'\}$
5.	$\{Vu{=}z \mid \varepsilon\}$	$\to \{u{=}\varepsilon \mid \varepsilon\},\{V\backslash z\}$	10.	$\{Vu{=}z \mid Xw\}$	$\to \{Vu{=}zX \mid w\},\{\}$ $(*)$

Fig. 3 The prefix unification algorithm for intuitionistic logic

unifiers. In the worst-case, the number of unifiers grows exponentially with the length of the prefixes p and q. To solve a *set* of prefix equations $\bar{E} = \{p_1 = p_1, \ldots, q_n = t_q\}$, the equations in $\bar{E}$ are solved one after the other and each calculated unifier is applied to the remaining prefix equations in $\bar{E}$.

For example, for the prefix equation $\{a_1 V_2 V_3 = a_1 a_3\}$, there are the two possible derivations $\{a_1 V_2 V_3 = \varepsilon | a_1 a_3\}, \{\} \xrightarrow{3.} \{V_2 V_3 = \varepsilon | a_3\}, \{\} \xrightarrow{6.} \{V_3 = \varepsilon | a_3\},\ \{V_2 \backslash \varepsilon\} \xrightarrow{10.} \{V_3 = a_3 | \varepsilon\}, \{V_2 \backslash \varepsilon\} \xrightarrow{5.} \{\varepsilon = \varepsilon | \varepsilon\}, \{V_2 \backslash \varepsilon,\ V_3 \backslash a_3\}$ and $\{a_1 V_2 V_3 = \varepsilon | a_1 a_3\}, \{\} \xrightarrow{3.} \{V_2 V_3 = \varepsilon | a_3\}, \{\} \xrightarrow{10.} \{V_2 V_3 = a_3 | \varepsilon\}, \{\} \xrightarrow{5.} \{V_3 = \varepsilon | \varepsilon\}, \{V_2 \backslash a_3\} \xrightarrow{5.} \{\varepsilon = \varepsilon | \varepsilon\}, \{V_2 \backslash a_3,\ V_3 \backslash \varepsilon\}$, yielding the most general unifiers $\{V_2 \backslash \varepsilon,\ V_3 \backslash a_3\}$ and $\{V_2 \backslash a_3,\ V_3 \backslash \varepsilon\}$.

4.2 Modal Logics

For modal logic the classical sequent calculus is extended by rules for the modal operators $\Box$ and $\Diamond$. For example, the additional *modal rules* of the *modal sequent calculus* [70] for the modal logic T are

$$\frac{\Gamma, F \vdash \Delta}{\Gamma, \Box F \vdash \Delta}\ \Box\text{-}\mathit{left}\,, \quad \frac{\Gamma \vdash F, \Delta}{\Gamma \vdash \Diamond F, \Delta}\ \Diamond\text{-}\mathit{right}\,, \quad \frac{\Gamma_{(\Box)} \vdash F, \Delta_{(\Diamond)}}{\Gamma \vdash \Box F, \Delta}\ \Box\text{-}\mathit{right}\,, \quad \frac{\Gamma_{(\Box)}, F \vdash \Delta_{(\Diamond)}}{\Gamma, \Diamond F \vdash \Delta}\ \Diamond\text{-}\mathit{left}$$

with $\Gamma_{(\Box)} := \{G \,|\, \Box G \in \Gamma\}$ and $\Delta_{(\Diamond)} := \{G \,|\, \Diamond G \in \Delta\}$. When the rules $\Box$-*right* or $\Diamond$-*left* are applied from bottom to top during the proof search, all formulae that are not of the form $\Box G$ or $\Diamond G$, respectively, are deleted from the sets $\Gamma_{(\Box)}$ and $\Delta_{(\Diamond)}$ in the premise. As these formulae might be necessary to complete the proof, the application of the modal rules need to be controlled. Again, a prefix is assigned to every subformula G of a given formula F. This *prefix* is a string over an alphabet $\nu \cup \Pi$, in which ν is a set of *prefix variables* and Π is a set of *prefix constants*. Prefix variables and constants represent applications of the rules $\Box$-*left* or $\Diamond$-*right*, and $\Box$-*right* or $\Diamond$-*left*, respectively [69, 70].

Proof-theoretically, a prefix of a subformula G captures the modal context of G and specifies the sequence of modal rules that have to be applied analytically in order to obtain G in the sequent. Semantically, a prefix denotes a specific world in a model [27, 70]. Prefixes of literals that form an axiom in the sequent calculus need to denote the same world, hence, they need to unify under a *modal substitution* σ_M that maps elements of ν to strings over $\nu \cup \Pi$. A connection $\{L_1 : p_1, L_2 : p_2\}$ is

Table 3 The definition of the prefixed matrix for modal logics

Type	$F^{pol} : p$	$M(F^{pol} : p)$
atomic	$A^0 : p$	$\{\{A^0 : p\}\}$
	$A^1 : p$	$\{\{A^1 : p\}\}$
α	$(\neg G)^0 : p$	$M(G^1 : p)$
	$(\neg G)^1 : p$	$M(G^0 : p)$
	$(G{\wedge}H)^1 : p$	$M(G^1 : p) \cup M(H^1 : p)$
	$(G{\vee}H)^0 : p$	$M(G^0 : p) \cup M(H^0 : p)$
	$(G{\Rightarrow}H)^0{:}p$	$M(G^1 : p) \cup M(H^0 : p)$
ν	$(\Box G)^1 : p$	$M(G^1 : pV^*)$
	$(\Diamond G)^0 : p$	$M(G^0 : pV^*)$
β	$(G{\wedge}H)^0 : p$	$M(G^0 : p) \cup_\beta M(H^0 : p)$
	$(G{\vee}H)^1 : p$	$M(G^1 : p) \cup_\beta M(H^1 : p)$
	$(G{\Rightarrow}H)^1{:}p$	$M(G^0{:}p) \cup_\beta M(H^1 : p)$
γ	$(\forall xG)^1 : p$	$M(G[x\backslash x^*]^1 : p)$
	$(\exists xG)^0 : p$	$M(G[x\backslash x^*]^0 : p)$
δ	$(\forall xG)^0 : p$	$M(G[x\backslash t^*]^0 : p)$
	$(\exists xG)^1 : p$	$M(G[x\backslash t^*]^1 : p)$
π	$(\Box G)^0 : p$	$M(G^0 : pa^*)$
	$(\Diamond G)^1 : p$	$M(G^1 : pa^*)$

σ-complementary for a combined substitution $\sigma := (\sigma_Q, \sigma_M)$ iff $\sigma_Q(L_1) = \sigma_Q(\overline{L_2})$ and $\sigma_M(p_1) = \sigma_M(p_2)$. An additional *domain condition* specifies if *constant*, *cumulative*, or *varying domains* are considered [70].

The *skolemization* technique is extended to modal logic by introducing a Skolem term also for the prefix constants [48]. This integrates the *irreflexivity test* into the term and prefix unification. The *prefixed matrix* $M(F^{pol}{:}p)$ of a *prefixed formula* $F^{pol}{:}p$ is a set of *prefixed clauses*, in which *pol* is a polarity and *p* is a prefix, and is defined inductively according to the Table 3 [48]. The definitions of $\cup_\beta$, x^*, and t^* are identical to the ones used for intuitionistic logic. V^* is a new prefix variable, a^* is a prefix constant of the form $f^*(x_1, \ldots, x_n)$, in which f^* is a new function symbol and $x_1, \ldots, x_n$ are all free term and prefix variables in $(\Box G)^0 : p$ or $(\Diamond G)^1 : p$. The *modal matrix* $M(F)$ of a modal formula F is the prefixed matrix $M(F^0 : \varepsilon)$, in which ε is the empty string.

For example, the modal matrix of the formula

$$\Box\, man(Plato) \;\Rightarrow\; \Diamond\, man(Plato) \tag{12}$$

is

$$\{\{man(Plato)^1 : V_1\}, \{man(Plato)^0 : V_2\}\} \tag{13}$$

1. $\{\varepsilon = \varepsilon \mid \varepsilon\}$	$\rightarrow \{\}, \{\}$	6. $\{au = \varepsilon \mid Vw\}$	$\rightarrow \{Vw = a \mid u\}, \{\}$
2. $\{\varepsilon = \varepsilon \mid Xw\}$	$\rightarrow \{Xw = \varepsilon \mid \varepsilon\}, \{\}$	7. $\{Vu = \varepsilon \mid \bar{V}w\}$	$\rightarrow \{w = V \mid u\}, \{\bar{V} \backslash \varepsilon\}$
3. $\{Vu = \varepsilon \mid \varepsilon\}$	$\rightarrow \{u = \varepsilon \mid \varepsilon\}, \{V \backslash \varepsilon\}$	8. $\{Vu = X \mid w\}$	$\rightarrow \{u = X \mid w\}, \{V \backslash \varepsilon\}$
4. $\{Xu = \varepsilon \mid Xw\}$	$\rightarrow \{u = \varepsilon \mid w\}, \{\}$	9. $\{Vu = X \mid w\}$	$\rightarrow \{u = \varepsilon \mid w\}, \{V \backslash X\}$
5. $\{\bar{V}u = \varepsilon \mid Xw\}$	$\rightarrow \{\bar{V}u = X \mid w\}, \{\}$	10. $\{au = V \mid w\}$	$\rightarrow \{u = \varepsilon \mid w\}, \{V \backslash a\}$

Fig. 4 The prefix unification algorithm for the modal logic T

in which V_1 and V_2 are prefix variables. Then,

$$\big[\,\big[\,\mathit{man(Plato)}^1 : V_1\,\big]\;\big[\,\mathit{man(Plato)}^0 : V_2\,\big]\,\big]$$

is a graphical modal connection proof of matrix 13 with $\sigma_M(V_1) = V_2$.

The core of the formal *clausal connection calculus for modal logic* [21, 48] is identical to the one for intuitionistic logic given in Fig. 2. The only difference to the intuitionistic calculus is the definition of the prefixes and the prefix unification algorithm. The clausal connection calculus for modal logic is *correct* and *complete*.

A *prefix unification algorithm* [48] is used to calculate the modal substitution σ_M. Depending on the modal logic, the *accessibility condition* has to be respected when calculating this substitution: for all $V \in \nu$: $|\sigma_M(V)| = 1$ for the modal logic D, $|\sigma_M(V)| \leq 1$ for the modal logic T; there is no restriction for the modal logics S4 and S5. The prefix unification for D is a simple pattern matching, for S4 the prefix unification for intuitionistic logic can be used, for S5 only the last character of each prefix (or ε if the prefix is empty) has to be unified. The prefix unification for T is specified by the *rewriting rules* given in Fig. 4 (with $\bar{V} \neq X$), which are applied in the same way as the ones for intuitionistic logic (see Sect. 4.1).

5 Implementing Connection Calculi

Several automated theorem provers for classical logic that are based on clausal connection calculi have been implemented so far, such as KoMeT [18], METEOR [5], PTTP [64], and SETHEO [39]. Because of their complexity, it would be a difficult — if not impossible — task to adapt these implementations to the non-classical connection calculi described in Sect. 4.

At first, this section presents a very compact PROLOG implementation of the clausal connection calculus for classical logic. Afterwards, this implementation is extended to intuitionistic and modal logics.

5.1 Classical Logic

leanCoP is an automated theorem prover for classical first-order logic [45, 46, 51]. It is a very compact PROLOG implementation of the clausal connection calculus described in Sect. 3.1. leanCoP 1.0 [51] essentially implements the basic clausal connection calculus shown in Fig. 1. The source code of the core prover is given in Fig. 5 (sound unification has to be used). PROLOG lists are used to represent sets and PROLOG terms are used to represent atomic formulae. PROLOG variables represent term variables, and "`-`" is used to mark literals that have polarity 1. For example, the matrix

$$\{\{man(Plato)^1\}, \{man(X)^0, mortal(X)^1\}, \{mortal(Plato)^0\}\}$$

is represented by the PROLOG list

```
[[-man(plato)],[man(X),-mortal(X)],[mortal(plato)]].
```

The prover is invoked by calling the predicate `prove(M,I)`, in which `M` is a matrix and `I` is a positive number. The predicate succeeds only if there is a connection proof for the matrix `M`, in which the size of the active path is smaller than `I`. The proof search starts by applying the start rule implemented in the first two lines. As a first optimization technique, the clause C_1 in the start rule of Fig. 1 can be restricted to *positive* clauses, i.e. clauses that contain only literals with polarity 0 [51]. For the above example this would be the clause $\{mortal(Plato)^0\}$. Afterwards, reduction and extension rules are repeatedly applied. These rules are implemented in the last four lines by the PROLOG predicate `prove(C,M,P,I)`, in which `C` is the subgoal clause, `M` is the matrix, `P` is the active path, and `I` is the path limit. The path limit is used to perform iterative deepening on the size of the active path, which is necessary for completeness. When the extension rule is applied, the proof search continues with the left premise before the right premise is considered. The axiom is implemented in the third line. The term substitution σ is stored implicitly by PROLOG.

leanCoP 1.0 already shows an impressive performance and proves some formulae not proven by more complex automated theorem provers [51]. As clause copies are restricted to *ground* clauses, leanCoP is also a decision procedure for determining the validity of *propositional* formulae.

leanCoP 2.0 integrates additional optimization techniques into the basic connection calculus [45, 46]. The source code of the core prover is shown in Fig. 6. *Lean PROLOG technology* is a technique that stores the clauses of the matrix in PROLOG's database. It integrates the main advantage of the "PROLOG technology" approach [64] into leanCoP by using PROLOG's fast indexing mechanism to quickly find connections. A *controlled iterative deepening* stops the proof search if the current path limit for the size of the active path is not exceeded (line 4). This yields a decision procedure for ground formulae and also allows for refuting some (not valid) first-order formulae. The *regularity* condition [38] restricts the proof search such that

```
prove(M,I) :- append(Q,[C|R],M), \+member(-_,C),
 append(Q,R,S), prove([!],[[-!|C]|S],[],I).
prove([],_,_,_).
prove([L|C],M,P,I) :- (-N=L; -L=N) -> (member(N,P);
 append(Q,[D|R],M), copy_term(D,E), append(A,[N|B],E),
 append(A,B,F), (D==E -> append(R,Q,S); length(P,K), K<I,
 append(R,[D|Q],S)), prove(F,S,[L|P],I)), prove(C,M,P,I).
```

Fig. 5 The source code of the leanCoP 1.0 core prover for classical logic

```
prove(I,S) :- \+member(scut,S) -> prove([-(#)],[],I,[],S) ;
    lit(#,C,_) -> prove(C,[-(#)],I,[],S).
prove(I,S) :- member(comp(L),S), I=L -> prove(1,[]) ;
    (member(comp(_),S);retract(p)) -> J is I+1, prove(J,S).
prove([],_,_,_,_).
prove([L|C],P,I,Q,S) :- \+ (member(A,[L|C]), member(B,P),
    A==B), (-N=L;-L=N) -> ( member(D,Q), L==D ;
    member(E,P), unify_with_occurs_check(E,N) ; lit(N,F,H),
    (H=g -> true ; length(P,K), K<I -> true ;
    \+p -> assert(p), fail), prove(F,[L|P],I,Q,S) ),
    (member(cut,S) -> ! ; true), prove(C,P,I,[L|Q],S).
```

Fig. 6 The source code of the leanCoP 2.0 core prover for classical logic

no literal occurs more than once in the active path (lines 6–7). The *lemmata* technique [38] reuses the subproof of a literal in order to solve the same literal on other branches (line 7). *Restricted backtracking* [46] cuts off alternative connections once the application of the reduction or extension rule has successfully solved a literal (line 11; see also Sect. 3.2). Backtracking over alternative start clauses can be cut off as well. A *definitional clausal form translation* is used in a preprocessing step to translate arbitrary first-order formulae into an equivalent clausal form by introducing definitions for certain subformulae [46]. Furthermore, leanCoP 2.0 uses a *fixed strategy scheduling*, i.e. the PROLOG core prover is consecutively invoked by a shell script with different strategies [45, 46]. See [46] for a more detailed explanation of the source code.

The core prover in Fig. 6 is invoked with `prove(1,S)`, where `S` is a strategy (see [45] for details) and the start limit for the size of the active path is 1. The predicate succeeds if there is a connection proof for the clauses stored in PROLOG's database. The full source code of the prover and the definitional clausal form translation are available on the leanCoP website at `http://www.leancop.de`.

The additional techniques improve the performance of leanCoP significantly, in particular for formulae containing many axioms [46]. Of the (non-clausal) formulae in the TPTP v3.7.0 problem library, leanCoP 2.0 proves (within 600 s) about 50% more formulae than leanCoP 1.0, about as many formulae as Prover9 [41], and about

30% less formulae than E [61]. The new definitional clausal form translation performs significantly better than those of E, SPASS, and TPTP [46].

nanoCoP [50] implements the non-clausal calculus described in Sect. 3.3. It proves more problems from the TPTP library than the core prover of leanCoP for both, the standard and the definitional translation into clausal form. Furthermore, the returned non-clausal proofs are on average about 30% shorter than the clausal proofs of leanCoP.

5.2 Intuitionistic Logic

ileanCoP is a prover for first-order intuitionistic logic [44, 45]. It is a compact PROLOG implementation of the clausal connection calculus for intuitionistic logic described in Sect. 4.1. ileanCoP extends the classical connection prover leanCoP by

a. prefixes that are added to the literals in the matrix in a preprocessing step,
b. a set of prefix equations that are collected during the proof search,
c. a set of term variables together with their prefixes in order to check the interaction condition, and
d. an additional prefix unification algorithm that unifies the prefixes of the literals in each connection.

The source code of the ileanCoP 1.2 core prover is shown in Fig. 7. The underlined code was added to the classical prover leanCoP 2.0; no other modifications were made. Prefixes are represented by PROLOG lists, e.g. the prefix $a_1 V_2 V_3$ is represented by the list `[a1,V2,V3]`. For example, the intuitionistic matrix

$$\{\{man(Plato)^1 : a_1 V_1\}, \{man(X)^0 : a_1 V_2\, a_2(X),\\ mortal(X)^1 : a_1 V_2 V_3\}, \{mortal(Plato)^0 : a_1 a_3\}\}$$

is represented by the PROLOG list

```
[[-man(plato):[a1,V1]], [man(X):[a1,V2,a2(X)],
  -mortal(X):[a1,V2,V3]], [mortal(plato):[a1,a3]].
```

In a preprocessing step the clauses of the intuitionistic matrix are written into PROLOG's database. Then, the prover is invoked with `prove(1,S)`, where `S` is a strategy (see [45] for details) and the start limit for the size of the active path is 1. The predicate succeeds if there is an intuitionistic connection proof for the clauses stored in PROLOG's database.

First, ileanCoP performs a classical proof search, which uses only a *weak* prefix unification (line 11 and line 12). After a classical proof is found, the prefixes of the literals in each connection are unified and the interaction condition is checked. To this end, the two predicates `prefix_unify` and `check_addco` are invoked (line 4). They implement the rewriting rules shown in Fig. 3 and require another 26

```
prove(I,S) :- ( \+member(scut,S) ->
    prove([(-(#)):(-[])],[],I,[],[Z,T],S) ;
    lit((#):_,G:C,_) -> prove(C,[(-(#)):(-[])],I,[],[Z,R],S),
    append(R,G,T) ), check_addco(T), prefix_unify(Z).
prove(I,S) :- member(comp(L),S), I=L -> prove(1,[]) ;
    (member(comp(_),S);retract(p)) -> J is I+1, prove(J,S).
prove([],_,_,_,[[],[]],_).
prove([L:U|C],P,I,Q,[Z,T],S) :- \+ (member(A,[L:U|C]),member(B,P),
    A==B), (-N=L;-L=N) -> ( member(D,Q), L:U==D, X=[], O=[] ;
    member(E:V,P), unify_with_occurs_check(E,N),
    \+ \+ prefix_unify([U=V]), X=[U=V], O=[] ;
    lit(N:V,M:F,H), \+ \+ prefix_unify([U=V]),
    (H=g -> true ; length(P,K), K<I -> true ;
    \+p -> assert(p), fail), prove(F,[L:U|P],I,Q,[W,R],S),
    X=[U=V|W], append(R,M,O) ), (member(cut,S) -> ! ; true),
    prove(C,P,I,[L:U|Q],[Y,J],S), append(X,Y,Z), append(J,O,T).
```

Fig. 7 The source code of the ileanCoP 1.2 and MleanCoP 1.2 core provers for intuitionistic and modal logics

lines of PROLOG code. If the prefix unification or the interaction condition fails, the search for alternative connections continues via backtracking. The substitutions σ_Q and σ_J are stored implicitly by PROLOG. The full source code is available on the ileanCoP website at `http://www.leancop.de/ileancop`.

Version 1.0 of ileanCoP is based on leanCoP 1.0 and implements only the basic calculus [44]. In ileanCoP 1.2 all additional optimization techniques used for classical logic described in Sect. 5.1 are integrated as well [45]. To this end, some of these techniques are adapted to the intuitionistic approach using prefixes. This includes regularity, lemmata, and the definitional clausal form translation. Other techniques, such as the lean PROLOG technology and restricted backtracking, can be applied directly without any modifications.

ileanCoP 1.2 proves significantly more formulae of the TPTP problem library and the ILTP problem library [55] than any other automated theorem prover for first-order intuitionistic logic [45]. Of the (non-clausal) formulae of the TPTP v3.3.0 problem library it proves (within 600 s) between 250 and 700% more formulae than the intuitionistic provers JProver, ileanTAP, ft, and ileanSeP [45]. It solves about 50% more formulae than ileanCoP 1.0, and proves significantly more formulae than Imogen [42]. Of the formulae of the TPTP v3.7.0 problem library, ileanCoP 1.2 proves a higher number of problems of certain problem classes than some *classical* provers [46], even though these classes contain formulae that are valid in classical but not in intuitionistic logic.

5.3 Modal Logics

MleanCoP is a prover for several first-order modal logics [48, 49]. It is a compact PROLOG implementation of the clausal connection calculi for modal logics, as described in Sect. 4.2. The source code of the MleanCoP 1.2 core prover is identical to the source code of ileanCoP 1.2 shown in Fig. 7. PROLOG lists are used to represent sets and prefixes. For example, the modal matrix

$$\{\{man(Plato)^1 : V_1\}, \{man(Plato)^0 : V_2\}\}$$

is represented by the PROLOG list

```
[[-man(plato):[V1]],[man(plato):[V2]]].
```

In a preprocessing step, the clauses of the modal matrix are written into PROLOG's database. First, MleanCoP performs a classical proof search using a *weak* prefix unification. After a classical proof is found, the prefixes of the literals in each connection are unified and the domain condition is checked. This is done by the two predicates `prefix_unify` and `domain_cond`. Depending on the chosen modal logic, the prefix unification algorithm has to respect different accessibility conditions. For example, for the modal logic T, the rewriting rules shown in Fig. 3 are used. For the modal logic S4, the code of the prefix unification for intuitionistic logic can be used. For D and S5, the prefix unification is a simple pattern matching. If the prefix unification or the domain condition fails, the search for alternative connections continues via backtracking. The substitutions σ_Q and σ_M are stored implicitly by PROLOG. The full source code is available on the MleanCoP website at `http://www.leancop.de/mleancop`.

As MleanCoP 1.2 is based on leanCoP 2.0, all additional optimization techniques used for classical logic described in Sect. 5.1 are integrated into this implementation as well [48]. This includes the lean PROLOG technology, regularity, lemmata, the definitional clausal form translation, and restricted backtracking. MleanCoP supports the constant, cumulative, and varying domain variants of the first-order modal logics D, T, S4, and S5.

MleanCoP 1.2 proves more formulae from the QMLTP v1.1 problem library [56] than any other prover for first-order modal logic, such as the provers LEO-II, Satallax, MleanSeP, or MleanTAP [21]. For the modal logic D, MleanCoP 1.2 proves (within 600 s) between 35 and 120% more problems than any of the other provers; for the modal logic T, between 25 and 85% more problems; for the modal logic S4, between 30 and 100% more problems, and for the modal logic S5, between 40 and 110% more problems. MleanCoP is also able to refute a large number of modal formulae that are not valid.

Version 1.3 of MleanCoP [49] contains additional enhancements, such as the support for heterogeneous multimodal logics, the output of a compact modal connection proof, support for the modal TPTP input syntax [56] and an improved strategy scheduling.

6 A Brief History and Perspectives

One of the fundamental achievements of logic is the discovery that truth can be demonstrated in a purely syntactic way. This means that any statement, represented as a formula F in some language, can be shown to be valid by purely syntactical means. All known methods for such a demonstration use syntactic rules of roughly the kind $F_1 \Rightarrow F_2$ and some termination criterion which, applied to a formula in the chain of demonstration, specifies whether the respective line in this chain can successfully stop at the point of the formula's occurrence.

When the idea of testing the validity of formulae in a mechanical way came up in the beginning of the last century, the most obvious way of choosing appropriate rules of this kind was to inverse the rules of the formal logical systems then known for deriving valid formulae. Even if the logic is restricted to fol, its formulae are rather complex syntactic constructs as are the rules of those systems. Hence this approach led to a rather complicated solution first realized by Prawitz [53] (for more historical details concerning the beginnings of ATP see Sect. 2 in [17]).

The problem with this solution was tractability — tractability from the point of view of human researchers and system developers, that is. Hence some kind of simplification was called for. It was Herbrand who first succeeded in reducing the problem of determining the validity of fol formulae to propositional ones [34]. This reduction is known as *Herbrand's Theorem*. Since propositional logic is much easier to handle for human researchers this opened a line of research additionally characterized as a confluent saturation method based on the resolution rule along with unification which to some extent hides first-order features.

When the second author, abbreviated as WB in the following, entered the field of ATP around 1970 as a trained logician, the Prawitz line was out of date and the Herbrand-resolution line was highly in vogue. This observation took him by surprise as expressed at the beginning of his first publication in ATP [11]: *"In the field of theorem proving in first-order logic almost all work is based on Herbrand's theorem. This is a surprising fact since from a logical point of view the most natural way … ."* Hence, for nearly a decade he tried to further develop the Prawitz line and at the same time to study the virtues and disadvantages of both lines in a comparative way, in order to reach a rational decision which line to follow in the future.

In the course of this research and after several publications WB, like Prawitz and others, realized that a core of ATP lies in the connection structure of the given formula (or of the set of clauses resulting from it), independent of which line of research is pursued. With this insight different approaches to ATP could be developed and analysed from a common viewpoint as demonstrated in the paper [13]. The paper was completed in 1978, published as "Bericht 79" in January 1979 and as a journal article in 1981 (submitted 1979). It already contains all the basic notions underlying the matrix characterization of logical validity, provides the basis for the connection method (CM) and features a number of different results.

In 1979 WB attended CADE-4 where Peter Andrews presented his paper [6] later published as [7]. This independently taken approach turned out to be very closely related to the one taken in the CM while using a different terminology (mating instead of spanning set of connections, etc.). It cites one of WB's papers while WB had not been aware of Andrews' work before this talk. Therefore, it is not yet cited in the report version of 1979, but then of course in the journal version [13]. Andrews' future work focussed more on higher-order logic while WB's work in ATP continued the line taken so far, producing [14] as well as the book [15] among numerous other publications including ones with several co-authors. It eventually resulted in the system SETHEO [39]. SETHEO in 1996 won first-place at the first international competition among theorem provers, the CADE ATP System Competition or CASC.

There is a close relative to the Prawitz line which we might call the Beth line, today known under the term tableaux (see e.g. [28]). The team realizing the implementation of SETHEO featured two members who by education were committed to thinking in terms of tableaux rather than of the CM. Hence, SETHEO was influenced by the CM and some of its features but cannot be called a proper CM-prover. The prover KoMeT [18] may be regarded a more authentical product in this respect; but its main developer unfortunately soon left the field thus terminating its development towards an internationally competitive system. Hence, the leanCoP prover family discussed in this chapter has become the first long-term project on the very basis of the CM.

Most researchers in ATP are still committed to the resolution approach in ATP. In fact, the two most successful systems in terms of the CASC competition are based on resolution. On first sight, this seems to be a good reason to regard resolution superior to its competitors. But the argument is not really convincing if a closer look is taken, as we will try to show now.

The high performance of resolution systems is to a large extent due to the following two reasons. First of all, resolution was designed intentionally as a *machine-oriented* inference rule which could be implemented relatively easily and in a way so that an extremely massive amount of inferences can be performed in a short time. Due to the resulting successes of those implementations many systems have been developed on its basis. So, secondly, sheer numbers of investments have made the systems ever more powerful. Winning a CASC competition may well be a consequence of these two peculiarities and does not necessarily say something in sufficient detail about the ultimate potential of the underlying proof method.

In fact, resolution in principle does suffer from serious inherent drawbacks. It operates on sets of clauses resulting from the original formula F to be proved. The formula is built from axioms, theorems, lemmas and the assertions and has exactly in this respect an information-rich structure, from which human mathematicians draw heavily as they search for a proof of the assertions. This structure is totally destroyed in the resolution context. In order to cope with this deficiency to some extent, strategies like set-of-support have been developed. While they are surely useful to some extent, they do not bring back in full the rich information contained in the original structure of F. Hence resolution inferences in present systems to a large extent are carried out in a rather blind way, a disadvantage compensated by brute force and sheer power due to the two peculiarities mentioned. But for proving truly hard theorems brute force does and will not suffice. Even Alan Robinson, the hero in ATP who has laid the basis for the resolution line, in his more recent work has convincingly argued into the same direction [59] but so far has largely remained unnoticed by the huge community of his followers who keep sticking exclusively to his earlier work.

For all these reasons we continue to be deeply convinced that, in contrast, the research path pursued on the basis of the matrix characterization and the CM in the long term will yield much better results. Let us therefore summarize here its ultimate vision of a future proof method which is based on Theorem 10.4 in [15]. The idea is to elaborate *within F* a deductively sufficient *skeleton* (cf. Definition 10.2 in [15]) which is characterized exclusively by the very syntactic items occurring in F and some relations defined for them (like the pairing relation defining connections, etc.). This goal has already been achieved by leanCoP for formulae in skolemized clausal form, and by nanoCoP for arbitrary formulae in skolemized (non-clausal) form, as discussed in Sect. 5 of the present chapter. The particular feature of *splitting by need* (Sect. 10 in [15]) has been studied extensively in [4, 33], but without focussing on an effective proof search guided by connections. There is the additional feature of integrating an alternative for skolemization, discussed in Sect. 8 of [15] (and in fact already introduced in Sect. 4 of [11]). It would not only preserve a given formula in its truly original structure but would also make the translation back to a more readable sequent proof significantly easier. Despite these advantages, this alternative for skolemization is widely unknown in the community.

Altogether a comprehensive calculus combining all these results along with a corresponding implementation is still missing because the way towards it is a truly hard one. Once it will be available additional strategies for guiding the proof search through the information given by the structural elements in F (axioms, theorems, etc.) may be explored for the first time within a competitive system. This will be possible since the calculus keeps the given formula in its original structure completely unchanged. This also opens a way to consider the realization and inclusion of Robinson's recent ideas referenced just before.

The line of research described here is unique and pursued so far by only a very few researchers worldwide. The reason for the lack of popularity also lies in the fact that the technical details are extremely challenging for anyone. There is a relationship with tableaux in that both lines originate from related formal systems of fol. But in contrast to tableaux which redundantly expand F to numerous subformulae according to the tableau rules, in our approach F is left completely untouched, instead accumulating information about its structure in the skeleton. Hence the Beth-line in terms of performance in principle cannot compete with our much more compact approach. But tableaux are much easier for humans to work with, thus explaining their continuing popularity.

The skeleton identified after a successful proof search represents a set of possible derivations in the underlying formal system of fol and in this sense is an extremely condensed abstraction from such a set of derivations. Any derivation of this set may easily be rolled out as soon as the skeleton has been found (see Corollary 10.6 in [15]). In this manner proofs become accessible to human understanding after they have been discovered by machine.

At some point in the last century people believed that resolution had an advantage in terms of complexity in the limit in comparison with our approach. But in the book [25] it has been shown that this in fact is not the case (see its Theorem 4.4.1), provided that one avoids repeating one and the same part of a connection proof redundantly over and over again. This is another feature which has to be cared for in a future system following our approach. For further aspects concerning such a future system see also [16, 19].

7 Conclusion

Formal reasoning in classical and non-classical logics is a fundamental technique when developing provably correct software. Over the last decades, the implementation of proof calculi has made considerable advances and automated theorem provers are nowadays used in industrial applications. But whereas the development of efficient ATP systems for *classical* logic has made significant progress (See e.g. [60]), the development of ATP systems for many important *non-classical* logics is still in its infancy. This is in particular true for *first-order* intuitionistic logic and *first-order* modal logics. Whereas the *time complexity* of determining whether a given *propositional* formula F is valid in classical logic is *co-NP-complete* [22], it is already *PSPACE-complete* [63] for intuitionistic or the (standard) modal logics (except for the modal logic S5, which is *co-NP*-complete [70]). Proof search in these non-classical logics is considerably more difficult than for first-order classical logic and, hence, only a few implementations of ATP systems for these logics exist to date.

This chapter provides a summary of research work on proof calculi and efficient implementations for classical and non-classical logics that has been carried out during the last ten years. All these calculi and implementations are based on the *connection method*. In particular, they are based on the matrix characterization of logical validity and operate in a goal-oriented connection-driven manner. As a result of this research work, three efficient automated theorem provers for first-order classical, first-order intuitionistic, and first-order modal logic are now available. All implemented ATP systems are elegant and very compact implementations based on uniform clausal connection calculi for classical, intuitionistic, and modal logics. Different non-classical logics are specified by prefixes in the clausal matrix and additional prefix unification algorithms. The minimal PROLOG source code of the core theorem provers consists of between only 11 and 16 lines.

leanCoP is one of the strongest theorem provers for first-order *classical logic*. It has won several prizes at CASC, the yearly competition for fully automated ATP systems, such as the “Best Newcomer” award (leanCoP 2.0) [65] and the SUMO reasoning prize (leanCoP-SInE) [67] and was winner of the first arithmetic division (leanCoP-Ω) [68]. It is currently the most efficient theorem prover based on a connection calculus. leanCoP includes well-known techniques, such as regularity, lemmata, and some novel optimization techniques, such as lean PROLOG technology, an optimized definitional clausal form translation, and restricted backtracking. The definitional clausal form translation works significantly better than other well-known translations [46]. Restricted backtracking reduces the search space in connection calculi significantly, in particular for formulae containing many axioms. Indeed, restricted backtracking turned out to be the single most effective technique for pruning the search space in connection calculi.

The clausal connection calculus for classical logics was adapted to first-order *intuitionistic logic* and several first-order *modal logics*. To this end, an intuitionistic matrix and a modal matrix were defined, which add prefixes to the clausal matrix. The skolemization technique is extended to prefixes, hence, clause copies are made by simply renaming all term and prefix variables. The additionally required prefix unification algorithm is specified by a small set of rewriting rules and depends on the particular logic. ileanCoP extends leanCoP to first-order intuitionistic logic by adding prefixes to literals and integrating an intuitionistic prefix unification algorithm. It is currently the most efficient automated theorem prover for first-order intuitionistic logic. MleanCoP extends leanCoP to the first-order modal logics D, T, S4, and S5. To this end the definition of prefixes and the prefix unification algorithm is modified and adapted, whereas the core prover already used for ileanCoP remains unchanged. Experimental results indicate that the performance of MleanCoP is better than that of any other existing theorem prover for first-order modal logic.

In summary, the use of an additional prefix unification during the proof search for non-classical logics resembles the use of term unification in first-order logic:

$$\begin{array}{rcll}\textit{first-order}\text{ logic} & = & \textit{propositional}\text{ logic} & + \ \textit{term}\text{ unification}\ , \\ \textit{intuitionistic/modal}\text{ logic} & = & \textit{classical}\text{ logic} & + \ \textit{prefix}\text{ unification}\ .\end{array}$$

By capturing the intuitionistic and modal contents of formulae in prefixes, most optimization techniques, such as the definitional clausal form translation and restricted backtracking, can be used for intuitionistic and modal logic as well.

The *non-clausal* connection calculus is a generalization of the clausal connection calculus. To this end, a formal definition of non-clausal matrices is given, the extension rule is slightly modified and a decomposition rule is added. In contrast to existing approaches, clause copies are added carefully and dynamically during the proof search. The non-clausal connection calculus combines the advantages of more natural (non-clausal) sequent or tableau calculi with the goal-oriented property of connection calculi. The non-clausal connection calculus is implemented in the compact PROLOG theorem prover nanoCoP. nanoCoP not only returns more natural non-clausal proofs, but the proofs are also significantly shorter.

To sum up, the presented research work has provided sufficient evidence to support the assertion that connection calculi are a solid basis for efficiently automating formal reasoning in classical and non-classical logics. They have been implemented carefully in a very compact and elegant way. Whereas the resulting performance is similar or even superior to that of existing — significantly more complex — ATP systems, the correctness of the concise code of the core provers can be checked much more easily. Hence, such implementations can not only serve as tools for constructing *provably correct software*, but they themselves follow an approach that ensures that *they are* provably correct software. An observation that was made 25 years ago by Hoare [35]:

> I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

Acknowledgements We would like to thank several anonymous referees for their constructive comments which were helpful to improve the text. Our thanks are also due to the editors Jonathan Bowen, Michael Hinchey and Ernst-Ruediger Olderog for the organization of the ProCoS Workshop in 2015, for compiling this volume and for inviting us to both projects.

References

1. http://www.scientificamerican.com/article/pogue-5-most-embarrassing-software-bugs-in-history/. Accessed 15 June 2016
2. https://en.wikipedia.org/wiki/2015_Seville_Airbus_A400M_Atlas_crash. Accessed 15 June 2016
3. https://en.wikipedia.org/wiki/List_of_failed_and_overbudget_custom_software_projects. Accessed 15 June 2016
4. Antonsen, R., Waaler, A.: Liberalized variable splitting. J. Autom. Reason. **38**, 3–30 (2007)
5. Astrachan, O., Loveland, D.: METEORs: high performance theorem provers using model elimination. In: Bledsoe, W., Boyer, S. (eds.) Automated Reasoning: Essays in Honor of Woody Bledsoe, pp. 31–60. Kluwer, Amsterdam (1991)
6. Andrews, P.B.: General matings. In: Joyner, W.H. (ed.) Fourth Workshop on Automated Deduction, pp. 19–25 (1979)
7. Andrews, P.B.: Theorem proving via general matings. J. ACM **28**, 193–214 (1981)
8. Barwise, J.: An introduction to first-order logic. In: Barwise, J. (ed.) Handbook of Mathematical Logic, pp. 5–46. North-Holland, Amsterdam (1977)
9. Blackburn, P., van Bentham, J., Wolter, F.: Handbook of Modal Logic. Elsevier, Amsterdam (2006)
10. Baumgartner, P., Eisinger, N., Furbach, U.: A confluent connection calculus. In: Hölldobler, S. (ed.) Intellectics and Computational Logic. Applied Logic Series 19, pp. 3–26. Kluwer, Dordrecht (2000)
11. Bibel, W.: An approach to a systematic theorem proving procedure in first-order logic. Computing **12**, 43–55 (1974)
12. Bibel, W.: Syntax-directed, semantics-supported program synthesis. Artificial Intelligence **14**, 243–261 (1980)
13. Bibel, W.: On matrices with connections. J. ACM **28**, 633–645 (1981)
14. Bibel, W.: Matings in matrices. Commun. ACM **26**, 844–852 (1983)
15. Bibel, W.: Automated Theorem Proving, 2nd edn. Vieweg, Braunschweig (1987)
16. Bibel, W.: Research perspectives for logic and deduction. In: Stock, O., Schaerf, M. (eds.) Reasoning. Action, and Interaction in AI Theories and Systems - Essays dedicated to Luigia Carlucci Aiello, LNAI 4155, pp. 25–43. Springer, Berlin (2006)
17. Bibel, W.: Early history and perspectives of automated deduction. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNAI 4667, pp. 2–18. Springer, Berlin (2007)
18. Bibel, W., Brüning, S., Egly, U., Rath, T.: KoMeT In: Bundy, A. (ed.) CADE-12. LNAI 814, pp. 783–787. Springer, Heidelberg (1994)
19. Bibel, W., Otten, J.: From schütte's formal system to modern automated deduction. In: Kahle, R., Rathjen, M. (eds.), The Legacy of Kurt Schütte. Springer, London, to appear

20. Brandt, C., Otten, J., Kreitz, C., Bibel, W.: Specifying and verifying organizational security properties in first-order logic. In: Siegler, S., Wasser, N. (eds.) Verification, Induction, Termination Analysis. LNAI 6463, pp. 38–53. Springer, Heidelberg (2010)
21. Benzmüller, C., Otten, J., Raths, T.: Implementing and evaluating provers for first-order modal logics. In: De Raedt, L., et al. (eds.) 20th European Conference on Artificial Intelligence (ECAI 2012), pp. 163–168. IOS Press, Amsterdam (2012)
22. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing, pp. 151–158. ACM, New York (1971)
23. van Dalen, D.: Intuitionistic logic. In: Goble, L. (ed.) The Blackwell Guide to Philosophical Logic, pp. 224–257. Blackwell, Oxford (2001)
24. Deville, Y.: Logic Programming, Systematic Program Development. Addison-Wesley, Wokingham (1990)
25. Eder, E.: Relative Complexities of First Order Calculi. Vieweg, Braunschweig (1992)
26. Fisher, K.: HACMS: high assurance cyber military systems. In: Proceedings of the 2012 ACM Conference on High Integrity Language Technoloby, pp. 51–52. ACM, New York (2012)
27. Fitting, M.: Proof Methods for Modal and Intuitionistic Logic. D. Reidel, Dordrecht (1983)
28. Fitting, M.: First-Order Logic and Automated Theorem Proving, 2nd edn. Springer, Heidelberg (1996)
29. Gentzen, G.: Untersuchungen über das logische Schließen. Math. Z. **39**(176–210), 405–431 (1935)
30. Goel, S., Hunt, W.A., Kaufmann, M.: Engineering a formal, executable x86 ISA simulator for software verification. In: Bowen, J.P., Hinchey, M., Olderog, E.-R. (eds.) Provably Correct Systems. Springer, London (2016)
31. Hähnle, R.: Tableaux and related methods. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 100–178. Elsevier, Amsterdam (2001)
32. Hähnle, R., Murray, N.V., Rosenthal, E.: Linearity and regularity with negation normal form. Theor. Comput. Sci. **328**, 325–354 (2004)
33. Hansen, C.: A Variable Splitting Theorem Prover. University of Oslo (2012)
34. Herbrand, J.J.: Recherches sur la théorie de la démonstration. Travaux Soc. Sciences et Lettres Varsovie, Cl. 3 Mathem. Phys. (1930)
35. Hoare, C.A.R.: The emperor's old clothes. Commun. ACM **24**, 75–83 (1981)
36. Klein, G., Elphinstone, K., Heiser, G., Adronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: SeL4: formal verification of an OS kernel. In: Proceedings of the 22nd ACM SIGOPS, pp. 207–220. ACM, New York (2009)
37. Lee, S.-J., Plaisted, D.: Eliminating duplicates with the hyper-linking strategy. J. Autom. Reason. **9**, 25–42 (1992)
38. Letz, R., Stenz, G.: Model elimination and connection Tableau procedures. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 2015–2114. Elsevier, Amsterdam (2001)
39. Letz, R., Schumann, J., Bayerl, S., Bibel, W.: SETHEO: a high-performance theorem prover. J. Autom. Reason. **8**, 183–212 (1992)
40. Loveland, D.: Mechanical theorem proving by model elimination. J. ACM **15**, 236–251 (1968)
41. McCune, W.: Release of Prover9. Mile High Conference on Quasigroups, Loops and Nonassociative Systems. Technical report, Denver (2005)
42. McLaughlin, S., Pfenning, F.: Efficient intuitionistic theorem proving with the polarized inverse method. In: Schmidt, R.A. (ed.) CADE-22. LNCS 5663, pp. 230–244. Springer, Heidelberg (2009)
43. Moore, J.S.: Computing verified machine address bounds during symbolic simulation of code. In: Bowen, J.P., Hinchey, M., Olderog, E.-R. (eds.) Provably Correct Systems. Springer, London (2016)
44. Otten, J.: Clausal connection-based theorem proving in intuitionistic first-order logic. In: Beckert, B. (ed.) TABLEAUX 2005. LNAI 3702, pp. 245–261. Springer, Heidelberg (2005)

45. Otten, J.: *leanCoP* 2.0 and *ileanCoP* 1.2: high performance lean theorem proving in classical and intuitionistic logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS 5195, pp. 283–291. Springer, Heidelberg (2008)
46. Otten, J.: Restricting backtracking in connection calculi. AI Commun. **23**, 159–182 (2010)
47. Otten, J.: A Non-clausal Connection Calculus. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011. LNAI 6793, pp. 226–241. Springer, Heidelberg (2011)
48. Otten, J.: Implementing connection calculi for first-order modal logics. In: Ternovska, E., Korovin, K., Schulz, S. (eds.), 9th International Workshop on the Implementation of Logics (IWIL 2012), EPiC, EasyChair, vol. 22, pp. 18–32 (2012)
49. Otten, J.: *MleanCoP*: a connection prover for first-order modal logic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNAI 8562, pp. 269–276. Springer, Heidelberg (2014)
50. Otten, J.: *nanoCoP*: a non-clausal connection prover. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016, LNAI 9706. Springer, Heidelberg (2016)
51. Otten, J., Bibel, W.: *leanCoP*: lean connection-based theorem proving. J. Symb. Comput. **36**, 139–161 (2003)
52. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**, 293–304 (1986)
53. Prawitz, D.: A proof procedure with matrix reduction. In: Laudet, M., et al. (eds.) Symposium on Automatic Demonstration. Lecture Notes in Mathem, pp. 207–214. Springer, Berlin (1970)
54. Rautenberg, W.: A Concise Introduction to Mathematical Logic. Springer, Heidelberg (2010)
55. Raths, T., Otten, J., Kreitz, C.: The ILTP problem library for intuitionistic logic. J. Autom. Reason. **38**, 261–271 (2007)
56. Raths, T., Otten, J.: The QMLTP problem library for first-order modal logics. In: Gramlich, B., et al. (eds.) IJCAR 2012. LNAI 7364, pp. 454–461. Springer, Heidelberg (2012)
57. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**, 23–41 (1965)
58. Ray, S.: Scalable Techniques for Formal Verification. Springer, Heidelberg (2010)
59. Robinson, J.A.: Proof = guarantee + explanation. In: Hölldobler, S. (ed.) Intellectics and Computational Logic. Applied Logic Series 19, pp. 277–294. Kluwer, Dordrecht (2000)
60. Robinson, J.A., Voronkov, A.: Handbook of Automated Reasoning. Elsevier, Amsterdam (2001)
61. Schulz, S.: E - a brainiac theorem prover. AI Commun. **15**, 111–126 (2002)
62. Smullyan, R.M.: First-Order Logic. Springer, Heidelberg (1968)
63. Statman, R.: Intuitionistic propositional logic is polynomial-space complete. Theoret. Comput. Sci. **9**, 67–72 (1979)
64. Stickel, M.: A Prolog technology theorem prover: implementation by an extended Prolog compiler. J. Autom. Reason. **4**, 353–380 (1988)
65. Sutcliffe, G.: The CADE-21 automated theorem proving system competition. AI Commun. **21**, 71–81 (2008)
66. Sutcliffe, G.: The TPTP problem library and associated infrastructure: the FOF and CNF parts, v3.5.0. J. Autom. Reason. **43**, 337–362 (2009)
67. Sutcliffe, G.: The CADE-22 automated theorem proving system competition - CASC-22. AI Commun. **23**, 47–59 (2010)
68. Sutcliffe, G.: The 5th IJCAR automated theorem proving system competition - CASC-J5. AI Commun. **24**, 75–89 (2011)
69. Waaler, A.: Connections in nonclassical logics. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1487–1578. Elsevier, Amsterdam (2001)
70. Wallen, L.A.: Automated Deduction in Nonclassical Logics. MIT Press, Cambridge (1990)

Part VI
Run-Time Assertion Checking

Run-Time Deadlock Detection

Frank S. de Boer and Stijn de Gouw

Abstract This chapter reports research that is partly funded by the EU project FP7-610582 ENVISAGE. It describes a method for detecting at run-time deadlock in both multi-threaded Java programs and systems of concurrent objects. The method is based on attribute grammars for specifying properties of message sequences. For multi-threaded Java programs we focus on the actual tool-development which extends the run-time checking of assertions. For concurrent objects which communicate via asynchronous message passing and synchronize on futures which store the return values, we present the underlying theory and sketch its implementation.

1 Introduction

As early as in 1949, Alan Turing suggested the use of assertions in a talk "Checking a Large Routine" at Cambridge, for specifying and proving program correctness. This use of assertions in the logical specification of the mathematical relations between the values of the program variables was further developed by Floyd in inductive assertion networks and by Hoare in a programming logic. Furthermore, checking assertions at run-time is an important practical method for finding bugs.

In [1] we enhanced run-time assertion checking with attribute grammars [2] for describing properties of *histories*, e.g., sequences of method calls and returns. This supports strict programming to interfaces because it allows for interface specifications abstracting from the state as represented by the program variables. In [3] we extended this approach to multi-threaded Java programs which avoids interference problems in a natural manner. In this chapter we show how we can express, and

F.S. de Boer (✉) · S. de Gouw
CWI, Amsterdam, The Netherlands
e-mail: F.S.de.Boer@cwi.nl

S. de Gouw
Open University, Amsterdam, The Netherlands

F.S. de Boer
Leiden University, Leiden, The Netherlands

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_10

detect at run-time, deadlock in both multi-threaded Java programs and Actor-based programs (as for example introduced in [4]) by means of attribute grammars.

Related Work In [3] we showed how our approach to run-time assertion checking can be extended to multi-threaded Java programs while avoiding in a natural manner interference problems. As an example of the generality of our approach, we showed in [3] how to express deadlocks in multi-threaded Java programs. In this chapter we detail the actual implementation of this application to deadlock detection in multi-threaded Java programs. We further show how to express and detect deadlock arising in systems of concurrent objects which communicate via *asynchronous method calls* and so-called *futures* which store the return values (as described in [5]).

One of the main related works [6] describes how to detect deadlock *potentials* in multi-threaded programs which may give rise to *false positives* and *false negatives*. We however focus on detecting *actual* deadlocks at run-time.

There exist a variety of *static* techniques for deadlock analysis. Such techniques analyze the source code *without* executing it and aim at establishing absence of deadlock in *all* executions or finding a counter-example, i.e., a deadlocking computation. In general the computational complexity of the algorithms underlying these techniques is a major obstacle to their application to large software systems. Furthermore, their application in general requires certain abstractions which give rise to imprecision. For example, in [7] a CFL-reachability analysis[1] for deadlock in multi-threaded Java programs is introduced which is based on a finite abstraction provided by the underlying call-graphs. As another example, in [8] Dynamic Push down Networks (DPNs) are introduced as an abstract model for parallel programs with (recursive) procedures and dynamic process creation. Further, in [9–11] different techniques for the deadlock analysis of systems of concurrent objects are introduced based on a variety of abstractions, e.g., abstract descriptions of method's behaviours.

2 The Framework

This section briefly summarizes the use of attribute grammars in run-time verification as presented in [1]. We use the interface of the Java `BufferedReader` (Fig. 1) as a running example to explain the basic modeling concepts.

Communication View A communication view is a (possibly partial) mapping which associates a name to each event. Partiality makes it possible to filter out irrelevant events and event names are convenient in referring to events.

Suppose we wish to formalize the following property of the `BufferedReader`:

> The `BufferedReader` may only be closed by the same object which created it, and reads may only occur between the creation and closing of the `BufferedReader`.

This property must hold for the local history of all instances. The intuitive idea behind this property is that the object that opened (created) the buffer "owns" it,

[1]Here CFL stands for "Context Free Language".

```
interface BufferedReader {
  void close();
  void mark(int readAheadLimit);
  boolean markSupported();
  int read();
  int read(char[] cbuf, int off, int len);
  String readLine();
  boolean ready();
  void reset();
  long skip(long n);
}
```

Fig. 1 Methods of the BufferedReader interface

```
local view BReaderView grammar BReader.g
specifies java.util.BufferedReader {
  BufferedReader(Reader in) open,
  BufferedReader(Reader in, int sz) open,
  call void close() close,
  call int read() read,
  call int read(char[] cbuf, int off, int len) read
}
```

Fig. 2 Communication view of a BufferedReader

and is as such responsible for closing it, but it may pass the buffer on to clients that can read from it (so in particular, reads are allowed by multiple other objects). The communication view in Fig. 2 selects the relevant events and associates them with intuitive names: $open$, $read$ and $close$.

All return and call events not listed in the view are filtered. Note how the view identifies two different events (calls to the overloaded read methods) by giving them the same name $read$. Though the above communication view contains only provided methods (those listed in the `BufferedReader` interface), required methods (e.g. methods of other interfaces or classes) are also supported. Since messages to such methods are sent to objects of a different class (or interface), one must include the appropriate type explicitly in the method signature. For example, if we additionally include the following event in the view:

```
call void C.m() out
```

then all call-messages to the method `m` of class `C` sent by a `BufferedReader` are selected and named out. In general, incoming messages received by an object correspond to calls of provided methods and returns of required methods. Outgoing messages sent by an object correspond to calls of required methods and returns of provided methods. Incoming call-messages of local histories never involve static methods, as such methods do not have a callee.

Local communication views, such as the one in Fig. 2, select messages sent and received by *a single object* of a particular class, indicated by ‘specifies java.util. BufferedReader’. In contrast, global communication views select messages sent and received by *any* object during the execution of the Java program. This is useful to specify global properties of a program. In addition to instance methods, calls and returns of static methods can also be selected in global views.

In contrast to interfaces of the programming language, communication views can contain constructors, required methods, static methods (in global views) and can distinguish methods based on return type or method modifiers such as ‘static’, or ‘public’. The following features are supported: constructors, inheritance, dynamic binding, overloading, static methods, access modifiers. In addition to these features, in Sect. 4 we add support for multi-threading. We associate a grammar to each view. The grammar keyword, followed by a file name indicates the file containing the grammar associated to the view (i.e. Fig. 2 refers to the grammar in the file `BReader.g`). The next section discusses grammars in detail.

Grammars The context-free grammar underlying the attribute grammar in Fig. 3 generates the valid histories for `BufferedReader`, describing the prefix closure of sequences of the terminals ‘open’, ‘read’ and ‘close’ as given by the regular expression (open read* close). In general, the event names form the terminal symbols of the grammar, whereas the non-terminal symbols specify the structure of valid sequences of events. In our approach, a communication history is valid if and only if it and all its prefixes are generated by the grammar.

We extend the grammar with attributes for specification of the *data-flow* of the valid histories. Each terminal symbol has *built-in* attributes named `caller`, `callee` and the parameter names for respectively the object identities of the caller, callee and actual parameters. Terminals corresponding to method returns additionally have an attribute `result` containing the return value. Non-terminals have *user-defined* attributes to define data properties of sequences of terminals. We extend the attribute grammar with assertions to specify properties of attributes. For example, in the attribute grammar in Fig. 3 a user-defined synthesized attribute ‘c’ for the non-terminal ‘C’ is defined to store the identity of the object which closed the `BufferedReader` (and is `null` if the reader was not closed yet). Synthesized

```
S ::= open R  {assert (open.caller == null || open.caller == R.c ||
                       R.c == null);}
    |  ϵ
R ::= read R1 (R.c = R1.c;)
    |  C       (R.c = C.c;)
C ::= close C1 (C.c = C1.caller;)
    |  close   (C.c = close.caller;)
    |  ϵ       (C.c = null;)
```

Fig. 3 Attribute grammar which specifies that ‘read’ may only be called in between ‘open’ and ‘close’, and the reader may only be closed by the object which opened it

attributes define the attribute values of the non-terminals on the left-hand side of each grammar production, thus the ‘c’ attribute is not set in the productions of the start symbol ‘S’.

The assertion allows only those histories in which the object that opened (created) the reader is also the object that closed it. Throughout the chapter the start symbol in any grammar is named ‘S’. For clarity, attribute definitions are written between parentheses ‘(’ and ‘)’ whereas assertions over these attributes are surrounded by braces ‘{’ and ‘}’. We use subscripts to distinguish different occurrences of the same non-terminal, i.e., in the grammar below C and C_1 are different occurrences of the non-terminal C.

Assertions can be placed at any position in a production rule and are evaluated at the position they were written. Note that assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions are placed directly after the terminal. This is in fact a generalization of traditional pre- and post-conditions for methods as used in design-by-contract: a single terminal ‘call-m’ can appear in multiple productions, each of which is followed by a different assertion. Hence different preconditions (or post-conditions) can be used for the same method, depending on the context (grammar production) in which the event corresponding to the method call/return appears.

Figure 4 shows a parse tree of the sequence of terminals ‘open read read close’, where the caller of open and close is the same object o_1, but the second read operation is triggered by another object o_2. Terminals - corresponding read, open or close events - are shown as rectangles in the parse tree with a built-in attribute ‘caller’. A circle

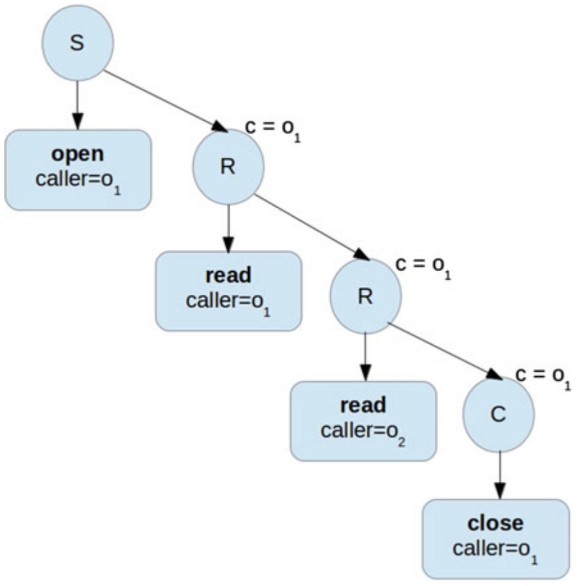


Fig. 4 Parse tree of ‘open read read close’

denotes a non-terminal, with a user-defined attribute ‘c’ for the non-terminals C and R to store the object that last closed it.

3 Deadlock Detection for Concurrent Objects

In this section we discuss the run-time detection of deadlock in systems of concurrent objects as described in [5]. Such systems consist of objects which communicate via *asynchronous method calls* and so-called *futures* which store the return values. An asynchronous method call $v = e!m(\bar{e})$ (where $\bar{e}$ denotes the sequence of actual parameters of the call of method m of the called object denoted by e, and where v denotes a future), generates a corresponding *closure* which is stored in the process queue of the callee. A closure consists of a (sequential) statement, e.g., the body of a method, and a local environment specifying the values of the local variables (including the formal parameters). The future variable v stores a reference to the return value (as such it can be passed around). The operation v.get *blocks* the current active closure till the return value has been generated. On the other hand, the operation v.await *suspends* the current active closure by storing it in the process queue till the return value has been generated. This allows so-called *cooperative* scheduling of another closure for execution. All objects are executing their active closures concurrently and fully encapsulate their local data. See Fig. 5 for the formal syntax with the following non-terminals: T for types, P for programs, L for classes, M for methods, sr for statements which return a value, s for any other statement, v for fields and local variables, f for fields, and, finally, x for local variables. By $\overline{X}$, where X denotes a sequence of symbols, we denote a sequence of X's. Types include class names C and types $\uparrow T$ of a future reference to a return value of type T. A program P consists of a sequence of class definitions L which supports class inheritance. A class definition consists of a sequence of method definitions M. A method is defined by its return type, the types of the formal parameters, and its body which terminates in a return statement. We abstract from the syntax of the side-effect free expressions e. The main statements of interest are side-effect free assignments $v = e$ to either a field f or a local variable x, object creation $v = \texttt{new}\ C()$, asynchronous method calls $v = v!m(\bar{e})$ and statements $v = v.\texttt{get}$ and $v.\texttt{await}$ which involve polling a future, as described informally above. We assume distinguished local variables `this` and

$$
\begin{array}{lll@{\qquad}lll}
T & ::= & C \mid\ !T \mid \ldots & P & ::= & \overline{L}\ \{\overline{Tx}; sr\} \\
L & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{Tf}; \overline{M}\} & M & ::= & T\ m\ (\overline{Tx})\{\overline{Tx}; sr\} \\
sr & ::= & s; \texttt{return}\ e & s & ::= & v = e \mid \\
 & & & & & v = \texttt{new}\ C() \mid v = v!m(\bar{e}) \mid v = v.\texttt{get} \\
 & & & & & v.\texttt{await}\ \mid \ldots \\
v & ::= & f \mid x & & &
\end{array}
$$

Fig. 5 The language syntax. Variables v are fields (f) or local variables (x), and C is a class name

```
class Service {
        Sensor sensor; Proxy proxy;
        Service(int val) {
                sensor = new Sensor; proxy = new Proxy(val);
        }
        void subscribe(Client cl) { proxy!add(cl) }
        void process() {
          while (true) {
                !Event fut = sensor!detectEvent();
                proxy!publish(fut);
                await fut?;}
        }
}

class Proxy {
        List<Clients> myClients; Proxy nextProxy;
        Event ev; int limit;
        Proxy(int k) {
                 limit = k; myClients = new List(); nextProxy = null;
        }
        void add(Client cl) {
                if myClients.length < limit { myClients.add(cl); }
                else { if nextProxy == null nextProxy = new Proxy(limit);
                        nextProxy.add(cl); }
        }
        void publish(!Event fut) {
                await fut?;
                if nextProxy != null { nextProxy!publish(fut); }
                ev = fut.get();
                for Client client : myClients { client!signal(ev); }
        }
}
```

Fig. 6 Publisher-subscriber pattern

dest which denote the executing object and the future dest uniquely identifying the corresponding method invocation.

Figure 6 presents a publisher-subscriber pattern which is taken from [5] and (quoting [5]) “wherein an event observed by a sensor is published to objects subscribed to a service. To avoid bottlenecks when publishing an event, the service delegates to a chain of proxy objects, where each proxy object informs both the next proxy and up to a specified limit of subscribing clients. We assume these classes exist: Sensor with method detectEvent, Client with method signal, and a list parametric in type T, with method add.”

The operational semantics is defined by a transition relation between global configurations (γ, δ, θ), where γ is the set of active closures, δ the set of suspended closures, and θ represents the (global) heap. A global heap assigns a local state to both the existing objects and futures. The local state $\theta(o)$ of an object o is an assign-

(CALL)

$$\frac{r \notin \mathrm{dom}(\theta) \quad \theta_\tau(v) = o \quad c = cl(C, m, o, r, \theta_\tau(\bar{e}))}{(\gamma \cup \{(\tau, u = v!m(\bar{e}); sr)\}, \delta, \theta) \rightarrow (\gamma \cup \{(\tau, u = r; sr)\}, \delta \cup \{c\}, \theta[r \mapsto \perp])}$$

(AWAIT1)

$$\frac{\theta_\tau(v) \neq \perp}{(\gamma \cup \{(\tau, v.\mathtt{await}; sr)\}, \delta, \theta) \rightarrow (\gamma \cup \{(\tau, sr)\}, \delta, \theta)}$$

(AWAIT2)

$$\frac{\theta_\tau(v) = \perp}{(\gamma \cup \{(\tau, v.\mathtt{await}; sr)\}, \delta \cup \{c\}, \theta) \rightarrow (\gamma, \delta \cup \{(\tau, v.\mathtt{await}; sr)\}, \theta)}$$

(SCHED)

$$\frac{c = (\tau, sr) \quad \forall(\tau', sr') \in \gamma : \tau'(\mathtt{this}) \neq \tau(\mathtt{this})}{(\gamma, \delta \cup \{c\}, \theta) \rightarrow (\gamma \cup \{c\}, \delta, \theta)}$$

RETURN

$$(\gamma \cup \{(\tau, \mathtt{return}\ e)\}, \delta, \theta) \rightarrow (\gamma, \delta, \theta[\tau(\mathtt{dest}) \mapsto \theta_\tau(e)])$$

Fig. 7 The operational semantics

ment of values to its fields, whereas the local state $\theta(r)$ of a future (reference) r is simply a value of the corresponding type or the value $\perp$ which stands for "undefined" (or "uninitialized"). A closure c is a pair (τ, sr), where τ is an assignment of values to the local variables.

Figure 7 gives the main operational rules. Here $\theta_\tau(e)$ denotes the value of the (side-effect free) expression e in the global heap θ and local environment τ, e.g., $\theta_\tau(x) = \tau(\mathtt{this})$, for every local variable x (including $\mathtt{this}$), and $\theta_\tau(f) = \theta(\tau(\mathtt{this}))(f)$, for every field f. For any sequence of expressions $\bar{e}$, we denote by $\theta_\tau(\bar{e})$ the corresponding sequence of values. Further, by $\theta[o.f \mapsto d]$ we denote the update of θ resulting from assigning the value d to the field f of object o, e,g., $\theta[o.f \mapsto d](o)(f) = d$. Similarly, by $\theta[r \mapsto d]$ we denote the update of θ resulting from assigning the return value d to the future reference r. The above notation is extended in the obvious manner to simultaneous updates. The rule CALL describes an asynchronous method call. It generates a fresh future reference r and a closure $cl(C, m, o, r, \theta_\tau(\bar{e}))$ which consists the body of the method (as defined in class C) and a local environment τ' such that $\tau'(\mathtt{this}) = o$, $\tau(\mathtt{dest}) = r$, and $\tau'(\bar{x}) = \theta_\tau(\bar{e})$, where $\bar{x}$ are the formal parameters. This closure is added to the set of suspended closures and the value of r is set to $\perp$. The rule AWAIT1 describes the continuation of the flow of control in case the polled future stores a returned value, whereas rule AWAIT2 describes suspending the active closure, in case the polled future is still undefined. The rule SCHED allows to schedule a suspended closure in case the object is idle, i.e., it has no active closure. This rule abstracts from the particular scheduling policy used and possible optimizations avoiding busy waiting, i.e., scheduling blocked await/get statements. The last rule RETURN describes the effect of the return statement in terms of the initialization of the corresponding future $\mathtt{dest}$.

Polling futures gives rise to a dependency relation between method invocations, e.g., a method invocation executing an await statement $v.\mathtt{await}$ depends on the execution of the method invocation uniquely identified by the future v to return a value. A cycle in this dependency relation between method invocations implies that we have a deadlock in the set of involved method invocations.

Definition 1 (*Deadlock*) Deadlock arises in a global configuration (γ, δ, θ) when there exist closures $c_i = (\tau_i, s_i; sr_i) \in \gamma \cup \delta$, where s_i either denotes an await statement v_i.`await` or a get operation $v = v_i$.`get`, such that $\tau_i(v_i) = \tau_{i\oplus 1}(\texttt{dest})$, $i = 1, \ldots, n$ ($\oplus$ here denotes addition modulo n).

In order to detect deadlock, the built-in attributes of events generated by asynchronous method calls denote, besides the caller, callee and the parameters, the generated future uniquely identifying the corresponding method invocation, which is denoted by the attribute name `dest`. The built-in attributes of events generated by return statements consist of the executing object (denoted by the attribute name `this`), the value returned (denoted by the name `val`) and the corresponding future (denoted by `dest`). The built-in attributes of events generated by await statements and assignments involving the `get` operation consist of the polled future (denoted by `fut`) and the future uniquely identifying the executing ("polling") method invocation (denoted by `dest`). In a (asynchronous) communication view we then can specify which synchronization events, i.e., `await`/`get` operations on futures which refer to the return value of a certain method, we want to observe by means of the specifications `await` *C.m* (and `get` *C.m*). By `await any` (`get any`) we refer to any `await` (`get`) operation.

Surprisingly, we can detect deadlock by *only* observing `await` and `get` operations, by means of the built-in attributes `fut` and `dest`, which denote the future which is polled and the future uniquely identifying the polling method invocation, respectively. This results in the following (global) communication view which maps every await/get operation on the same grammar token `poll` (Fig. 8).

The following grammar then generates, for each sequence of `poll` tokens, a corresponding graph of futures and checks absence of cycles (Fig. 9).

At run-time a given program instrumented with history updates which consist of adding a `poll` token just *before* every execution of an `await`/`get` operation then can be checked for absence of deadlock by simply parsing the history according to the above attribute grammar. Clearly, a deadlock will generate an assertion failure. It is less obvious that an assertion failure indeed corresponds with a deadlock. Note for

```
global view DeadlockMyProgram grammar deadlock.g {
   await any poll
   get any poll
}
```

Fig. 8 Global asynchronous communication view

Fig. 9 Attribute grammar for deadlock detection

$S ::=$ `poll` { `g.addEdge(poll.dest,poll.fut);` }
  | ϵ {`assert g.noCycle();`}

example that edges are *not* removed when a future is initialized. However, because futures are assigned to only once we can argue as follows. Let (γ, δ, θ) result from the execution of an active closure which generates an assertion failure caused by the addition of an edge (r, r') in the graph denoted by g. Let $r' = r_0, \dots, r_{n-1} = r$ be the nodes in g such that between r_i and $r_{i \oplus 1}$ ($\oplus$ here denotes addition modulo n) there exists an edge. We have to show that for $i = 0, \dots, n-1$ there exist closures $c_i = (\tau_i, sr_i) \in \gamma \cup \delta$ such that $\tau_i(\texttt{dest}) = r_i$ and the initial statement of sr_i involves an await or get operation on the future $r_{i \oplus 1}$. We show by induction that there exists such a closure c_i. For $i = n-1$ let $c_{n-1} = (\tau, sr)$ be the closure in γ such that $\tau(\texttt{dest}) = r$ and the initial statement of sr involves an await or get operation on the future r'. Next let $0 < i < n-1$ and $c_i = (\tau_i, sr_i)$ be the closure in $\gamma \cup \delta$ such that $\tau_i(\texttt{dest}) = r_i$ and the initial statement of sr_i involves an await or get operation on the future $r_{i \oplus 1}$. Let $c_{i-1} = (\tau_{i-1}, sr_{i-1})$ be the closure that resulted from the generation of the edge (r_{i-1}, r_i), i.e., $\tau_{i-1}(\texttt{dest}) = r_{i-1}$. Since $\tau_i(\texttt{dest}) = r_i$ and $c_i \in \gamma \cup \delta$ it follows from the operational semantics that $\theta(r_i) \neq \perp$. Therefore $c_{i-1} \in \gamma \cup \delta$ and the initial statement of sr_{i-1} involves an await or get operation on the future r_i.

4 Deadlock Detection for Multi-threaded Java Programs

Deadlocks in multi-threaded Java programs can arise from `Lock` objects, or from `synchronized` methods and statements. Deadlocks caused through using `Lock` objects can be detected in a straightforward manner by tracking calls to the `lock()` and `unlock()` methods, and do not require an extension to the framework introduced in the previous section. Thus we focus on deadlocks arising from `synchronized` methods. The program in Fig. 10 will be used as a running example. Depending on the scheduling, it can contain a deadlock: if the first thread starts executing `alphonse.bow(gaston)` but does not execute the call to `bowBack` before the second thread executes `gaston.bow(alphonse)`, the program deadlocks.

We specify different aspects of a multi-threaded program with the help of the following three perspectives:
Thread view: here we specify the behavior of each thread in isolation.
Object view: here we specify the behavior of objects individually.
Global view: here we specify global properties of a program.
All of the above views can be supported by a single formalism: attribute grammars extended with assertions, but the underlying history on which the grammar is evaluated differs between the various perspectives. The next subsection discusses multi-threaded events, and the required extensions to communication views to support the perspectives.

All grammars in this section are given in ANTLR [12] syntax: the input format of the underlying tool implementation (all grammars have been fully implemented and were used for run-time checking). The syntax of ANTLR grammars is close to Java: comments start with '//', and the actions (attribute definitions or assertions) in

```java
package nl.cwi.saga.deadlock;

import java.io.*;

public class Deadlock {
    public static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                        + " has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                        + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

Fig. 10 Example program with a potential deadlock. *Source* https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

the grammar are ordinary Java statements, surrounded by the braces ‘’ and ‘’. The left-hand side and right-hand side of a production are separated by a colon. ANTLR supports Extended BNF (EBNF): operators from regular expressions can be used in productions, such as the ‘*’ (zero or more repetitions) and ‘?’ (an optional symbol). Figure 12 shows an example grammar (the figure is discussed in more detailed in Sect. 4.2).

4.1 Multi-threaded Events

In a multi-threaded environment, events occur in different threads. Thus the first new ingredient compared to Sect. 2 is to keep track of the thread identity for each event. This is achieved with a new built-in attribute `Long threadId`. This attribute will be used in the deadlock detector to determine the events that wait on the completion of other events in a different thread.

In multi-threaded programs, due to scheduling and locking, there can be a delay between when a method is called, and when its body starts executing. For synchronized methods, a method call indicates that a lock was requested, whereas the start of the execution of a method body indicates that the lock was acquired successfully. To distinguish these two events, we introduce an ‘exec’ event, that indicates the start of execution of a method body (and thus implies acquisition of the lock). Returns of synchronized methods indicate the release of the lock.

4.2 Multi-threaded Perspectives

Thread View In the thread perspective, we specify the behavior of each thread in isolation. Each thread has its own dedicated history, and the grammar generates the set of valid histories of the thread. Semantically, such thread-local histories can be obtained from the global history by projection on the value of the `threadId` attribute (which, as mentioned above, stores the identity of the thread in which the event occurred).

We illustrate the thread view using the running example (Fig. 10). Figure 11 presents the corresponding communication view, introducing the grammar terminals “BOW” and “BOWBACK” for the corresponding events. Only events from implementations of the `Fork` interface with `synchronized` versions of `get` and `release` are selected.

We will specify that each person bows back to the same person that bowed to them. This intuitive property is formalized by the ANTLR grammar in Fig. 12. It specifies that each thread must first call `bow`, then `bowBack`, and (using an assertion) that the parameter of `bow` denotes the same object as the callee of `bowBack`.

```
thread view BowHistory grammar Bow.g {
    call public synchronized void
         Deadlock.Friend.bow(Friend bower) BOW,
    call public synchronized void
         Deadlock.Friend.bowBack(Friend bower) BOWBACK
}
```

Fig. 11 Communication view of `bow` and `bowBack`

```
grammar Bow;

///////////// HEADERS

/////////////////////// start ::= s EOF ///////////////////////////////
start : s EOF;

/////////////////////// s ::= BOW BOWBACK? | /\ //////////////////////
s : BOW
    (BOWBACK {assert $BOW.bower() == $BOWBACK.callee();})?
  | ;
```

Fig. 12 ANTLR attribute grammar specifying bowing behavior

Object View In the object view of a Java program, we specify the interactions of a single object with a communication view and corresponding grammar. The grammar generates the set of all valid traces of events that the object engages in. In a multi-threaded setting, several threads can be active (executing) in a single object, thus the object view is particularly useful for specifying (constrain) the order between events from different threads active in the same object. Intuitively, the local object histories can be obtained from the global history by projection on the values of the built-in attributes `caller` (for calls made by the object) and `callee` (for calls to the object).

For the bow-bowBack example, the object view is uninteresting: all interleavings/orderings between bows and bowBacks from different threads are allowed. A useful application of the object view is illustrated by the communication view in Fig. 2 and grammar in Fig. 3 in Sect. 2.

Global View The global view treats the Java program as a single entity that we wish to specify. The grammar generates the set of all valid *global* traces of the entire program. The user can specify the desired interleavings between events from different threads.

We use the global view for our deadlock detector. A thread blocks if it calls a synchronized method on an object that is already locked by another thread. The general idea is to build a directed "wait-for" graph to capture such dependencies between threads. A deadlock corresponds to a cycle in the wait-for graph.

In more detail, the nodes of the graph are thread id's, and there is an edge from t_1 to t_2 if t_1 calls a method on some object that is locked by t_2.

The view depicted in Fig. 13 selects the events relevant for deadlock detection. Note that we do not need to distinguish whether a certain event arose from `bow` or `bowBack`: the only information needed to identify deadlocks is which thread has requested/acquired/released the lock for which objects. Thus the calls to `bow` and `bowBack` are identified (mapped to the same terminal). The terminal "REQ" signifies requesting a lock, "ACQ" events are generated if a lock was acquired, and "REL" denotes the release of a lock.

```
global view DeadlockHistory grammar Deadlock.g {
    call public synchronized void
            Deadlock.Friend.bow (Deadlock.Friend bower) REQ,
    call public synchronized void
            Deadlock.Friend.bowBack(Deadlock.Friend bower) REQ,

    exec public synchronized void
            Deadlock.Friend.bow (Deadlock.Friend bower) ACQ,
    exec public synchronized void
            Deadlock.Friend.bowBack(Deadlock.Friend bower) ACQ,

    return public synchronized void
            Deadlock.Friend.bow (Deadlock.Friend bower) REL,
    return public synchronized void
            Deadlock.Friend.bowBack(Deadlock.Friend bower) REL
}
```

Fig. 13 Global communication view

Figure 14 shows an ANTLR attribute grammar that asserts no deadlock has occurred. To that end, a wait-for graph is built in the grammar productions with the help of two inherited attributes (syntactically in the ANTLR grammar, those are passed as parameters to the "s" non-terminal):

- An attribute `reqLock` of type `Map<Long, Object>` that maps a thread id (a `Long`) to the object for which it requested, but has not yet acquired the lock.
- An attribute `hasLock` of type `Map<Long, Map<Object, Integer> >`. Given a thread id and an object, this map returns the number of times the lock on that object has been acquired but not released by the thread.[2]

The attributes are updated in the grammar productions. In particular, the two maps are initialized to empty by the `start` production (line 6–7). Further:

- The production with the "REQ" terminal (line 12–16) signifies the request of a lock on the callee, correspondingly, in the grammar production we insert the thread identity and callee identity into the `reqLock` map.
- The production with terminal "ACQ" (line 18–31) signifies that the thread has successfully acquired the lock on the callee. Since the lock request for the callee is not pending anymore, the thread id is removed from the `reqLock` map. Additionally we increase the number of locks (due to re-entrance, a lock may have been acquired for that object already by the thread) that that thread has on the callee in the `hasLock` map.
- The "REL" terminal (line 33–42) signifies the release of a lock. In the grammar production we therefore decrease the number of locks that the thread has on the object. If the count becomes 0, the entry is removed.

[2]Due to re-entrance, locks in Java can be acquired more than once by the same thread.

```
grammar Deadlock;

///////////// HEADERS

//////////////////////// start ::= s EOF////////////////////////////////
start : s[new HashMap<Long, Object>(),
         new HashMap<Long, Map<Object, Integer> >()]
       EOF;

//////////////////////// s ::= (REQ | ACQ | REL)* /////////////////////
s[Map<Long, Object> reqLock, Map<Long, Map<Object, Integer> > hasLock] :
    REQ
    {
            reqLock.put($REQ.threadId(), $REQ.callee());
    }
    s[reqLock,hasLock]

  | ACQ
    {
        reqLock.remove($ACQ.threadId());
        Map<Object, Integer> m = hasLock.get($ACQ.threadId());
        int newCnt = 1;
        if(m == null) {
            m = new HashMap<Object, Integer>();
            hasLock.put($ACQ.threadId(), m);
        } else if(m.get($ACQ.callee()) != null)
            newCnt = m.get($ACQ.callee())+1;
        m.put($ACQ.callee(), newCnt);
    }

    s[reqLock,hasLock]

  | REL
    {
        Map<Object, Integer> m = hasLock.get($REL.threadId());
        Integer cnt = m.get($REL.callee());
        if(cnt == 1)
            m.remove($REL.callee());
        else
            m.put($REL.callee(), cnt-1);
    }
    s[reqLock,hasLock]

  |
  {
    DirectedGraph<Long, DefaultEdge> g =
        new DefaultDirectedGraph<Long, DefaultEdge>(DefaultEdge.class);
    for(Long rl : reqLock.keySet() ) {
        for(Long hl : hasLock.keySet() ) {
            if(rl != hl && hasLock.get(hl).containsKey(reqLock.get(rl))) {
                g.addVertex(rl);
                g.addVertex(hl);
                g.addEdge(rl, hl);
            }
        }
    }

    CycleDetector<Long, DefaultEdge> d =
        new CycleDetector<Long, DefaultEdge>(g);
    assert !d.detectCycles();
  };
```

Fig. 14 ANTLR attribute grammar specifying deadlocks

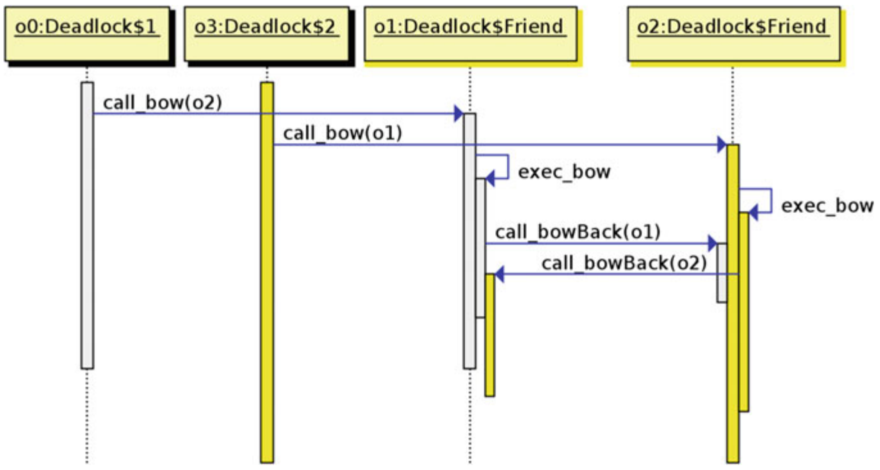


Fig. 15 Sequence diagram of a deadlocking executing of program Fig. 10

- The last production (the empty production, line 44–60) builds the wait-for graph: an edge is drawn from thread t_1 to thread t_2 if t_1 requests a lock owned by t_2. If $t_1 = t_2$ then t_1 has requested a lock that it already owns. In that case the lock can be acquired (no deadlock), thus we insert the edge only if $t_1 \neq t_2$. Since a cycle now corresponds to a deadlock, the assertion (line 60) is true if and only if there is no deadlock (Fig. 14).

As observed previously, there are schedulings for which the program in Fig. 10 deadlocks. We executed the program, checking for deadlocks using the given attribute grammar and encountered a deadlocking scenario. Our run-time checker prints certain information to aid debugging and isolate errors when an assertion fails or a parse error occurs: a stack trace that indicates the line of code where the error occurred, and a textual representation of the history that violated the specification. For example, the stack trace in Fig. 16 shows that execution failed at line 18 in the file Deadlock.java (Fig. 10).

That textual representation of the history can be visualised by the Quick Sequence Diagram Editor sdEdit. Figure 15 shows a visualization by sdEdit of a deadlocking trace. sdEdit gives each thread a color: in our case, gray(ish) and yellow. After the `exec_bow`-events, the gray thread owns the lock on o_1 and the yellow thread has the lock on o_2. With the two `call_bowBack`-events, the gray thread requests the lock for o_2 and the yellow thread requests the lock for o_1, thereby causing a deadlock.

```
java.lang.AssertionError
 at DeadlockParser.s(DeadlockParser.java:229)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.s(DeadlockParser.java:176)
 at DeadlockParser.s(DeadlockParser.java:176)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.s(DeadlockParser.java:146)
 at DeadlockParser.start(DeadlockParser.java:68)
 at DeadlockHistoryAspect$DeadlockHistory.parse
                                   (DeadlockHistoryAspect.java:502)
 at DeadlockHistoryAspect$DeadlockHistory.update
                                    (DeadlockHistoryAspect.java:603)
 at DeadlockHistoryAspect.ajc$before$DeadlockHistoryAspect$5$e8e2469d
                                    (DeadlockHistoryAspect.java:242)
 at Deadlock$Friend.bow(Deadlock.java:18)
 at Deadlock$2.run(Deadlock.java:36)
 at java.lang.Thread.run(Thread.java:745)
```

Fig. 16 Assertion failure in attribute grammar

5 Tool Architecture

For practical purposes, an important design goal of our run-time checker SAGA was to allow the use of up-to-date versions of the Java language. In particular, updates to the compilers should not break SAGA (in contrast, previous run-time checkers for JML specifications used a proprietary Java compiler which was not kept in sync with the Java language). The input of SAGA consists of a specification in the form of an attribute grammar with assertions, accompanied by a communication view. The output of SAGA is an AspectJ program for the generation of the events specified by the communication views (see [1]).

Choosing AspectJ as the output language of SAGA, allows the use of modern Java language versions, including the latest Java 8. AspectJ is tailored to the interception of events and as such is a most natural target language. An alternative approach would to instrument the program with a self-developed component of SAGA. But this is difficult because in general the instrumentation cannot be restricted locally to the methods that must be monitored. For example, since the identity of the caller is a built-in attribute of the grammar terminal, we cannot get away with instrumenting only the monitored methods, as one does not have access to the low-level stack in Java. Thus the identity of the callee is not accessible. This means that all call-sites should be instrumented.

However, the use of AspectJ raises certain challenges: we are now bound by limitations of Java. For a debugging tool such as a run-time checker, it would be convenient to have some control or access to various elements from the underlying execution platform, but this is often prohibited or even impossible in Java. For example: in a multi-threaded environment, during the evaluation of the specifications (i.e. the attribute grammar), another thread can potentially modify the heap. This would mean that different parts of the specification are evaluated in different states. Consider for example the assertion `assert x==x;`, where *x* is a field of an object. If after retrieving the value of the first occurrence of *x another* thread modifies *x* then the assertion may evaluate to false! This problem can be prevented if the run-time checker had control over the execution platform: it could then stall the other threads while a specification is evaluated. In [3] we discuss how we solved this without having control over the execution platform, and without stalling other threads (since this can cause a severe performance loss). A second implication arising from using AspectJ as target language is that to print an accurate sequence diagram, we must distinguish objects with different identities in the diagram. In Java, one can *test* objects for equality (using "=="), but in general there is no string that identifies each object uniquely (for example, the memory location for the object would qualify, but it is not accessible in Java). Thus SAGA generates a unique ID itself for each object appearing in the history.

Figure 17 shows an overview of the resulting tool architecture. It consists of an integration of four different components: a state-based assertion checker, a parser generator, a monitor to intercept events and a general tool for meta-programming. This architecture is further discussed in [3].

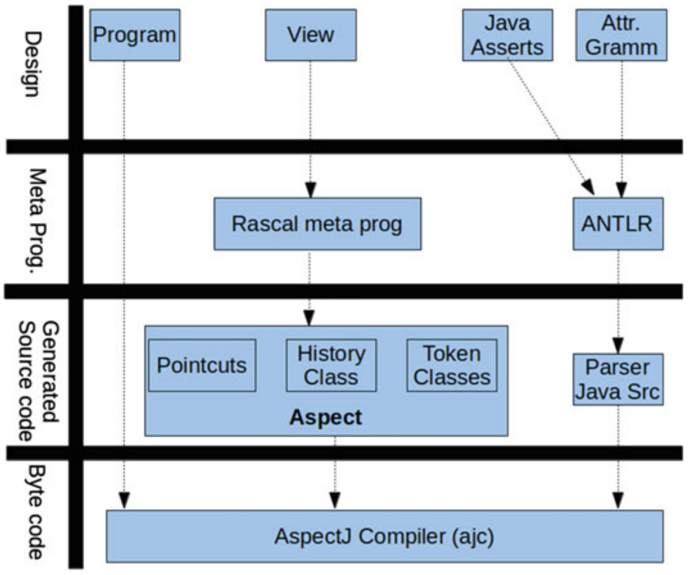


Fig. 17 SAGA tool architecture

6 Conclusion and Future Work

We discussed a method for the run-time detection of deadlock in both multi-threaded Java programs and systems of concurrent objects. The new version of SAGA which implements this method for multi-threaded Java programs can be obtained from https://github.com/cwi-swat/saga. Although we illustrated our framework for detecting deadlock for multi-threaded Java programs which use synchronized methods, general locks as provided in the package `java.util.concurrent.locks` can be handled just as easily by tracking the methods `lock`, `tryLock` and `unlock` in the communication view. What remains to be done is extending SAGA to deadlock detection of concurrent objects as described in this chapter. In general, future work will focus on further improving and extending the method by applying it to (industrial) case studies.

Acknowledgements We thank the anonymous reviewers for their most constructive and helpful comments.

References

1. de Boer, F.S., de Gouw, S., Johnsen, E.B., Kohn, A., Wong, P.Y.H.: Run-time assertion checking of data- and protocol-oriented properties of java programs: an industrial case study. Trans. Aspect-Oriented Softw. Dev. **11**, 1–26 (2014)
2. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**(2), 127–145 (1968)
3. de Boer, F.S., de Gouw, S.: Run-time checking multi-threaded java programs. In: 42nd International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM. Lecture Notes in Computer Science (2016)
4. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B., de Boer, F.S., Bonsangue, M.M. (eds.) Proceedings of 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
5. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, 24 March–1 April 2007, pp. 316–330 (2007)
6. Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., Stoller, S.D., Ur, S., Wang, L.: Detection of deadlock potentials in multithreaded programs. IBM J. Res. Dev. **54**(5), 3 (2010)
7. de Boer, F.S., Grabe, I.: Automated deadlock detection in synchronized reentrant multithreaded call-graphs. In: Proceedings of SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, 23–29 January 2010, pp. 200–211 (2010)
8. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: Proceedings of Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, 23–25 January 2011, pp. 199–213 (2011)
9. de Boer, F.S., Bravetti, M., Grabe, I., Lee, M.D., Steffen, M., Zavattaro, G.: A petri net based analysis of deadlocks for active objects and futures. In: Formal Aspects of Component Software,

9th International Symposium, FACS 2012, Mountain View, CA, USA, 12–14 September 2012. Revised Selected Papers, pp. 110–127 (2012)
10. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.Y.H.: Deadlock analysis of concurrent objects: Theory and practice. In: Proceedings of Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, 10–14 June 2013, pp. 394–411 (2013)
11. Giachino, E., Laneve, C.: Analysis of deadlocks in object groups. In: Proceedings of Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, 6–9 June 2011, pp. 168–182 (2011)
12. Parr, T.: The Definitive ANTLR Reference. Pragmatic Bookshelf (2007)

In-Circuit Assertions and Exceptions for Reconfigurable Hardware Design

Tim Todman and Wayne Luk

Abstract We present an approach to enable run-time, in-circuit assertions and exceptions in reconfigurable hardware designs. Static, compile-time checking, including formal verification, can catch many errors before a reconfigurable design is implemented. However, many other errors cannot be caught by static approaches, including those due to run-time data. Our approach allows users to add run-time assertions and exceptions to a design, giving multiple ways to handle run-time errors. We also allow imprecise assertions and exceptions, so that the origin of a failed assertion or raised exception is blurred. Users can take advantage of exception imprecision to trade performance for accurate location of errors. Our work includes a high-level approach to adding assertions and exceptions to a design, a concrete implementation for Maxeler streaming designs, and an evaluation. Results show low overhead for supporting assertions and exceptions in hardware design targeting FPGAs. For example, the cost of including assertions lies between 5% in lookup tables and 15% in Block RAMs in addition to the area used by the original design, due to logic used to implement assertion conditions, and buffers used to store assertion results. Furthermore, imprecision gives immediate benefits and up to 48% speedup over precise exceptions.

1 Introduction

As the size of reconfigurable hardware devices increases, they are used to implement increasingly large and complex designs. This leads to a challenge: verification, ensuring that designs implement their intended behaviour. There are many approaches to static, compile-time checking of designs, including formal verification, but static approaches cannot in general hope to catch all errors that can occur at run-time, particularly those caused by run-time input data.

T. Todman (✉) · W. Luk
Department of Computing, Imperial College London, 180 Queen's Gate,
London SW7 2AZ, UK
e-mail: timothy.todman@imperial.ac.uk

W. Luk
e-mail: w.luk@imperial.ac.uk

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_11

Traditionally, simulation is used to catch run-time errors, but designs are now so large that simulation cannot hope to catch them all. Assertion-based verification is increasingly popular; examples include Property Specification Language (PSL) [1] and System Verilog Assertions (SVA) [2]. In-circuit assertions [3] detect errors in hardware and report them to software. We propose in-circuit exceptions, which handle errors in the circuit where they are detected.

We define an *assertion* as any run-time Boolean expression which, when false, indicates an error of some kind, such as an input value out of range, or an intermediate result that will cause overflow. An *exception* is part of the control or data path that runs only when a corresponding assertion is false; if no assertions are false, no exception paths are active.

Assertions and exceptions separate error-handling code from normal operation, when no errors have been detected. Other language constructs could be used, but separating normal and error-handling code makes both easier to reason about. Assertions used in development may be removed for deployment; some criticize this as like "a sailing enthusiast who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea" [4].

For reconfigurable hardware, designs with more or fewer assertions and exceptions can be swapped at runtime. For example, many assertions about input data may be removed if processing a known-good set of inputs. Conversely, a design with more area devoted to assertions and exceptions could be configured if more than a set number of failed assertions or exceptions occur in a previous batch of input data. Furthermore, since reconfigurable hardware designs rarely fill the entire reconfigurable device, the unpopulated area can be filled with monitoring hardware, such as assertions and exceptions. We have separately developed an approach which reuses such *spare* resources for monitoring hardware, such as in-circuit assertions; since the monitors are added post place-and-route, the timing of the original design is preserved [5].

Our approach allows users to choose the *imprecision* of assertions and exceptions: imprecision means bounded inaccuracy in the reporting of where, or when, failed assertions and raised exceptions occurred. Zero imprecision means the reporting is accurate; more imprecision means more uncertainty about the exact code location or data input causing the error. Since area not spent on supporting assertions and exceptions can be used to increase performance, by increasing the degree of parallelization or pipelining, user programs controlling the reconfigurable hardware can make run-time tradeoffs between performance (fewer, lower-precision exceptions) and fidelity (more, higher-precision exceptions). Increasing imprecision also reduces the bandwidth between hardware and software.

Furthermore, in-circuit assertions and exceptions may be reused between different designs: for example, straightforward and optimized versions of the same design can share several assertions which check they implement the specification, or check assumptions on inputs and outputs.

This work makes the following contributions:

- An high-level, tool-agnostic approach to enabling runtime assertions and exceptions in hardware designs, with a language of assertion conditions and user-customizable policies for actions when assertions are violated;
- An implementation of our high-level approach for Maxeler streaming hardware designs, showing how the high-level approach maps into streaming hardware;
- The use of user-customizable imprecision, to allow time and space to be traded for less precise reporting of where a failed assertion originated from, or when it occurred;
- An evaluation of our approach on a case study. Results show low overhead for supporting assertions and exceptions in hardware design targeting FPGAs. For example, the cost of including assertions lies between 5% in lookup tables and 15% in Block RAMs in addition to the area used by the original design, due to logic used to implement assertion conditions, and buffers used to store assertion results.

The rest of the chapter is organized as follows: the next section outlines related work. Section 3 describes our high-level approach to runtime assertions for reconfigurable hardware designs; Sect. 4 details the implementation for Maxeler designs; Sect. 5 evaluates the approach, while Sect. 6 concludes and outlines future work.

2 Background

Software assertions have a long history: first, as a means of manually checking programs, then as a means of formally proving correctness, finally as a means of run-time checking of errors that cannot be detected statically [6]. Assertions are part of the C standard and by default print a message on the console before aborting. C has no built-in support for exceptions, but can emulate them using calls to jump back to functions deeper in the stack. Some languages have extensive support for exceptions, notably Ada and Eiffel [7].

2.1 Hardware Exceptions

The IEEE754 standard for floating-point arithmetic [8] includes exceptions, recommending that exceptions be resumeable, allowing user programs to fix problems.

Exception handling in pipelined or out-of-order processors is difficult because exceptions from later instructions may occur before earlier instructions finish. Some approaches include: allowing imprecise exceptions only, imprecise exceptions for only some instructions, choice of high-performance mode with imprecise exceptions or lower performance mode with precise exceptions; the precise mode can be 10 times slower or worse [9].

2.2 *Hardware Debugging*

Debugging circuits can correct a design after deployment, whereas exceptions are included from the beginning. Hung and Wilton [10] monitor signals in FPGA (Field Programmable Gate Array) designs by reclaiming unused routing resources; conditions causing errors can be observed but not corrected in-place. For hardware designs targeting FPGAs, Graham et al. [11] propose techniques for instrumenting bitstreams that are used in debugging FPGA circuits, while Poulos et al. [12] introduce hardware and software techniques for FPGA functional debug that leverage the inherent reconfigurability of the FPGA fabric to reduce functional debugging time.

2.3 *Assertion-Based Verification*

This lets designers add assertions to their designs, written in Boolean and temporal logic [13]. Approaches include PSL [1] and SVA [2]. These approaches only apply to simulation, not real hardware, and only to hardware parts of designs, not corresponding host software. Assertion-based verification has been extended to in-circuit assertions by Curreri [3], who extended ANSI-C assertions to streaming FPGA designs. This approach may catch some bugs caused by mismatches between software and hardware. However, there is no exception mechanism; user programs cannot recover from exceptions in hardware, only report errors back to software. In-circuit exceptions need not be restricted to Boolean expressions: recent work has explored the use of statistical assertions to check the mean or variance of circuit signals [14].

2.4 *Formal Verification*

Formal verification of hardware and software has a long history, with academic approaches going back to Hoare [15], Floyd [16], and Dijkstra [17], efforts which led to runtime verification by assertions [18]. There has also been research on automatic assume guarantee algorithms for assertion-based formal verification [19].

3 High-Level Approach

We now describe our high-level approach to runtime assertions and exceptions for streaming hardware designs. The approach is tool-agnostic: it does not depend on any particular tool, but could adapt to several available streaming hardware design tools.

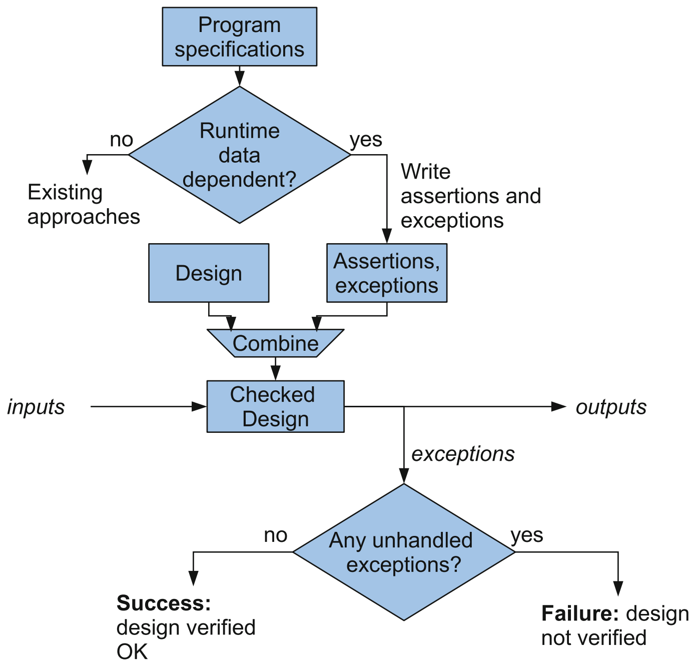


Fig. 1 Verification flow of our high-level approach: program specifications are partitioned into static and dynamic sets. Static specifications are handled by previous work. This work addresses dynamic specifications by translating them to assertions and exceptions, which are combined with an design to make a checked design. If, for all test inputs, there are no unhandled exceptions, the design is verified as correct (according to the dynamic specifications); otherwise, the unhandled exceptions can be used to debug the faulty design

We choose streaming hardware designs because they are increasingly used to implement reconfigurable hardware designs, particularly for high-performance applications. Much of our approach could also apply to other hardware design languages such as VHDL and Verilog.

The proposed verification flow (Fig. 1) starts with a design to verify and the properties to be verified. First, the user divides the properties into static or compile-time properties, and dynamic or run-time properties, dependent on run-time data. Second, the user separates the properties into assertions and exceptions; assertions encoding design properties, exceptions labelling error conditions. Static properties can be handled by existing static verification approaches. Third, the user writes run-time assertions to encode design assumptions which can only be checked at run-time, for example input variable ranges. For some exceptions, the user writes handlers to catch the exception and substitute a replacement value for the expression causing the exception: for example, an overflow exception might result in the value being clamped at the maximum value for that variable, resulting in a saturating arithmetic. Finally, the user runs the design including assertions and exceptions. If no assertions

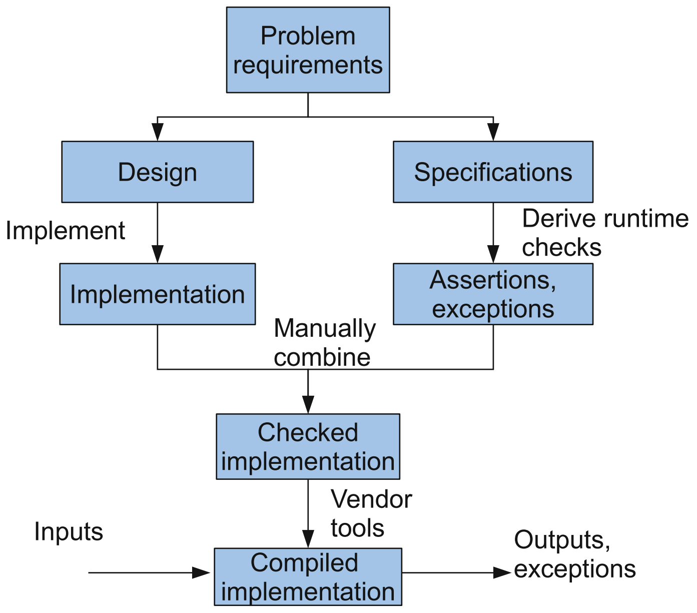


Fig. 2 Design flow of our high-level approach

are raised, and any exceptions are handled, the design is verified as correct for the input and assertions used.

Figure 2 shows the design flow of our high-level approach. This same design flow can be adapted to multiple concrete implementations for particular streaming hardware design descriptions; the next section shows an example. Our approach takes a streaming design language and augments it with features for run-time assertions and exceptions. Assertions are run-time Boolean conditions which, when false, cause an error. In software this often causes the program to terminate. In hardware, particularly streaming hardware, the design cannot simply terminate as the software host program may be waiting on the hardware for a fixed number of cycles. Our approach hence records the failed assertions on each cycle and returns these to the software host as extra outputs.

Multiple designs implement the same specifications, for example straightforward and optimized implementations. The same assertions and exceptions can be reused for both, to check requirements are met, saving design effort. Other assertions check design-specific properties.

Hardware exceptions differ from assertions in that they can be handled, meaning that a value is substituted for the expression which raised the exception. This allows designs to handle errors in place rather than relying on host software to fix the problem, potentially reducing bus traffic between software and hardware. Users can explore a tradeoff: handling more errors in hardware, costing more resources versus handling more errors in software, at a cost of more bandwidth required between hardware and software host.

Another use for assertions is *sanity checking*, a traditional programming technique where expressions which should always be true are asserted. Though this wastes resources (since the expression should always be true), it can reveal compiler bugs or designer conceptual errors or mistaken assumptions, e.g. wrong assumption about operator width [3].

Our approach allows assertion and exception *imprecision*: deliberately blurring the exact location of an exception either in time (which cycle the error occurs on, and hence which stream inputs caused it) or in space (which part of the program caused the error). Users can select the level of imprecision, defined as the number of stream inputs or program locations that could have caused the error; zero imprecision means the error is exactly located.

Imprecision can be thought of as data compression for error reporting: increasing imprecision reduces the size of error reporting data (and hence bandwidth) by a factor proportional to the imprecision.

To illustrate our approach, we introduce a simple language below which supports assertions and exceptions.

```
d = ...                                                1
  |'exception' ID';'                                   2
s = lval'=' expr                                       3
  |'if''(' e')' s'else' s                              4
  |'while''(' e')' s                                   5
  |'assert''(' e')'                                    6
e = e bop e                                            7
  | INT                                                8
  | FLOAT                                              9
  | ID                                                10
  |'(' e')'                                           11
  | uop e                                             12
  |'raise' ID                                         13
  |'try' e'with' ( ID'->' e )*                        14
lval= ID | ...                                        15
bop ='+' |'-' |'*' |'/' | ...                         16
uop ='+' |'-' |'~' | ...                              17
```

where d, s and e are declarations, statements and expressions respectively. Extensions for assertions and exceptions comprise:

- line 2: a declaration to declare possible exceptions in this program; only declared exceptions can be used;
- line 6: a statement to assert a condition: if false, an exception is raised;
- line 13: an expression to raise an exception;
- line 14: an expression to allow raised exceptions to be handled. Given an expression e, its result is e if no exceptions are raised in e, otherwise the optional list

of exception handlers is consulted. If a handler matches the raised expression, the corresponding value is the result of the expression, otherwise the exception propagates to the surrounding program.

The `assert` statement is directly taken from C99; many designers will already be familiar with this statement. Since C has no support for exceptions, we base our design on OCaml, which allows exceptions to be declared, raised and handled within both expressions and statements.

An informal semantics of our assertions and exceptions is:

- a failed assertion is recorded in a buffer showing which assertion failed, on which cycle;
- raising an undeclared exception is a compile-time error;
- raising an exception propagates it out to the enclosing expression;
- an exception raised within a try expression is matched against the list of handlers; if a handler matches, the corresponding expression results, otherwise the exception propagates to the surrounding expression;
- if an exception propagates to a statement, it is unhandled and recorded like a failed assertion.

3.1 Step-by-step Approach

We give a three-step approach to adding in-circuit assertions and exceptions to an existing high-level hardware synthesis tool:

(a) extend a high-level language targeting hardware implementation with language constructs for assertion and exception as shown above;
(b) provide circuit realisations of the assertions and exceptions, and during the hardware compilation process, instantiate them and link them to their uses in the design;
(c) include hardware and software Application Programming Interfaces with the extensions to allow hardware assertion and exceptions to propagate to software.

The next section shows how we implement these steps for a specific reconfigurable design technology.

4 Implementation: Maxeler Systems

We implement our high-level approach for Maxeler streaming systems. In the Maxeler system, users describe hardware designs as Java programs, using a Java class library and language extensions. When run, the programs build a dataflow graph of the design, compile the graph into an HDL (Hardware Description Language) implementation, and call FPGA vendor tools to compile the HDL into a bitstream. The design consists of a data path reading from one or more stream inputs, one per cycle, and producing one or more stream outputs, one per cycle. Scalar inputs are set once per run of the stream and are constant for each run. State machines or counters control the design.

Note that the Java programs, when run, generate the circuit. The designer can user Java assertions and exceptions in their program, but these will run at circuit generation time, not at circuit run time; thus they can only be used to debug or trap errors at circuit generation time. Our approach allows assertions and exceptions at circuit run time.

We systematically translate designs using Maxeler kernels extended with assertions and exceptions into regular Maxeler designs. Currently our translation is manual, but future work could automate it.

Following our three-step approach to extending a high-level hardware design tool with in-circuit assertions and exceptions, we extend the Maxeler system as follows:

4.1 Language Extensions for Runtime Assertions and Exceptions

We extend the Maxeler kernel description language, based on Java, with our high-level language features for runtime assertions and exceptions. We extend the grammar as follows:

```
d = ...
  |'__exception' ID';'
s = ...
  |'__try' s ('__catch''(' ID')' s )*
  |'__assert''(' e')'
  |'__raise' e';'
e = ...
  |'__try' e ( __when ID'->' e )*
  |'__raise' e
```

where existing grammar for declarations (`d`), statements (`s`), and expressions (`e`) is represented by ellipses (...). We allow exceptions to be raised and handled in both statements and expressions; this gives designers more choice about where to put error-handling code: one `__try ... __catch` block can handle any exceptions raised in the entire block.

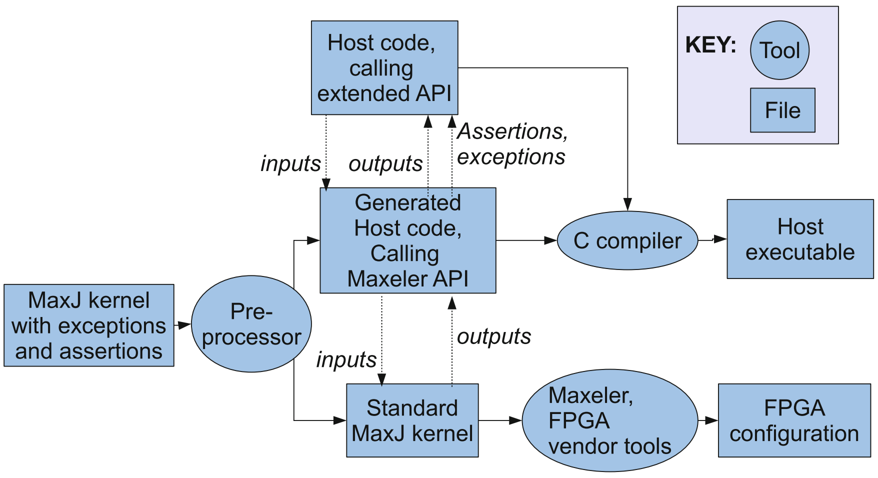

Host code, calling extended API
KEY:
Tool
File
inputs
outputs
Assertions, exceptions
Generated Host code, Calling Maxeler API
C compiler
Host executable
MaxJ kernel with exceptions and assertions
Pre-processor
inputs
outputs
Standard MaxJ kernel
Maxeler, FPGA vendor tools
FPGA configuration

Fig. 3 Design flow targeting Maxeler designs. The MaxJ code with exceptions and assertions is preprocessed, leading to standard MaxJ code with extra outputs representing assertions and exceptions, along with C code calling standard Maxeler APIs. Any failed assertions or raised exceptions in the hardware are mapped into failed assertions in the C code

Figure 3 shows the design flow for Maxeler systems. The user writes their design as a software program using our extended version of Maxeler's API (Application Programming Interface) for controlling a hardware design written in our extended version of Maxeler's MaxJ kernel description language (Maxeler's version of Java extended with syntax to ease the description of dataflow designs written using their class library). Our language extensions allow (a) assertions in hardware designs to be reflected into software designs; (b) exceptions to be declared, raised and handled in hardware designs. Unhandled exceptions similarly reflect into software.

4.2 Circuit Realizations of Assertions and Exceptions

For the Maxeler system with its streaming model, translation of assertions and exceptions to circuit realizations is straightforward. Each in-circuit assertion and raised exception is mapped to a separate single bit stream output. We systematically translate the design in the extended language into the same design in the standard language. Assertions are translated by connecting the appropriate output to the assertion condition, relying on the Maxeler tools to implement the condition. Raised exceptions are similarly implemented by connecting the output for that exception to the condition which govern whether that exception is raised.

A simple example code fragment:

```
1 __exception OutOfBounds;
2 ...
3 HWVar sum = (cond) ? (__raise OutOfBounds) :
    sum + x;
```

where the first line declares an exception, and the third raises that exception if `cond` is true. This is translated to:

```
1 HWVar condOut = cond;
2 HWVar sum = condOut ? sum : sum + x;
3 HWVar OutOfBounds1 = condOut;
4 io.output("OutOfBounds1", OutOfBounds1, hwBool
    ());
```

where:

- line 1 captures the result of the condition `cond` in a new local variable `condOut`;
- line 2 performs the assignment unless `cond` is true, meaning the exception would be raised;
- line 3 creates another new local variable corresponding to raising the exception; this is connected to the condition controlling whether the exception is raised, which was captured in `condOut`;
- finally, line 4 connects the exception to a new output of type `hwBool()`, meaning a single bit representing a Boolean value.

To implement imprecision, multiple exception or assertion outputs need to be combined together. Imprecision in space combines assertions from different parts of the design together; in Maxeler, this maps to a bitwise or operator. Imprecision in time combines the output same assertion from adjacent clock cycles; in Maxeler, this can be implemented using registers, multiplexers, and counters to control when assertion outputs are combined together.

4.3 Hardware and Software APIs for Assertions and Exceptions

Figure 4 shows how exceptions are supported by wrapping Maxeler hardware and software APIs. Each exception which can escape from the hardware becomes another streaming output, which must be passed using standard Maxeler APIs. In software, our tool adds a loop which performs a C software assertion for each exception output added.

4.4 Case Study

The following shows a basic C implementation of a 32-bit integer moving average filter, which we use as a basis for our experiments. The design is parameterised for

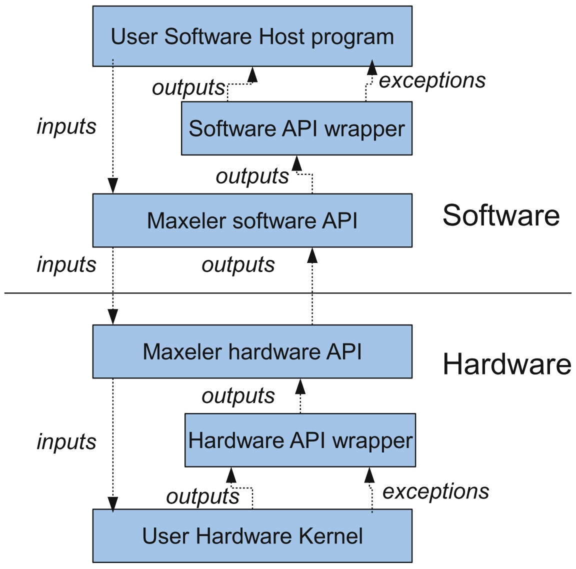


Fig. 4 Wrapping Maxeler hardware and software APIs

stream length N and filter radius W; we use arbitrary stream lengths and radius $W = 64$. This code reads from input array `inp` and writes to output array `outp`.

```
const size_t N=16*1024*1024;
int inp[N], outp[N];
for (i=0;i<N;++i) {
  sum=0;
  for (j=0;j<W;++j) {
    sum += inp[i-W/2+j];
  }
  outp[i] = sum/W;
}
```

For space reasons we omit code to stop reading outside the input array. A Maxeler implementation is:

```
__exception OutOfRange;//declare exception
HWVar inp = io.input("inp", hwInt(W));
__try {
  HWVar sum = constant.var(0);
  for (j=0;j<W;++j) {
    sum += stream.offset(inp,-(W/2)+j);
    if (sum<0) __raise OutOfRange;
  }
  __catch (OutOfRange) {
    sum = MAX;
}}
io.output("outp",sum/W,hwInt(W));
```

where:

- line 1 declares an exception;
- line 2 declares a stream input `inp` of 32-bit, unsigned integer type;
- lines 3 to 8 comprise a runtime exception-handling block: an OutOfRange exception raised in this block is handled by the corresponding catch block;
- line 4 declares a variable `sum` to store intermediate results;
- lines 5 to 8 implement the filter; this loop runs at compile-time (a fully-unrolled implementation);
- line 8 raises the OutOfRange exception if sum is negative (indicating overflow);
- lines 9 to 11 handle the exception from lines 4 to 8: if caught, sum is set to `MAX`;
- finally, line 12 declares output stream `outp`.

The following shows the C code generated by our example:

```
max_run(                                                1
  max_input(),                                          2
  max_output(),                                         3
  max_output(ex1),   // exceptions                      4
  max_runfor(N));                                       5
for (i = 0;i<NumExceptions-1;++i) {                     6
#line 7 "foo.maxj"                                      7
  assert(ex1[i] && "unhandled␣exception␣                8
     OutOfRange");
#line 20 "foo_generated.c"}                             9
```

where:

- lines 1 to 5 comprise a Maxeler API call to run a hardware kernel with inputs and outputs:
  - line 2 reads a C array as a user stream input to the kernel,
  - line 3 writes a C array as a user output from the kernel,
  - line 4 writes another C array of Boolean values, initialized to false, as exception outputs from the array;
  - line 5 gives the number of stream cycles to run;
- lines 6 to 10 comprise a C loop generating one assertion per raised exception or failed assertion, using a C preprocessor directive (`line`) to report from where in the hardware design file ("foo.maxj") the unhandled exception originated.

We augment Maxeler API calls interacting with the hardware to read back assertions and exception outputs, and generate one C assertion for each failed hardware assertion or unhandled exception. While C does not support exceptions, our approach could adapt to languages which do, so unhandled hardware exceptions lead to software exceptions.

Unlike C, several other languages do have native support for exception handling. Although our tool does not yet support other languages, it would be possible to modify the loop to throw software exceptions into the host program, for example in C++:

```
for (i = 0;i<NumExceptions-1;++i) {
  if (ex1[i]) {
#line 7"foo.maxj"
    throw std::runtime_error("sum>0, line 20");
#line 20 "foo_generated.c"
  }
}
```

where the loop throws one software exception for each unhandled hardware exception, in this case an object of the standard `runtime_error` class.

5 Evaluation

We evaluate using a moving average filter as a case study; though simple, similar tradeoffs in terms of area versus speed, and number of exceptions and assertions are needed in larger designs. Experiments measure the cost (reconfigurable hardware resources) to add assertions. Assertion imprecision results show how the user can trade exception imprecision for reconfigurable hardware resources and communications bandwidth.

5.1 Experimental Setup

Hardware is compiled using Maxeler MaxCompiler version 2012.1 and Xilinx ISE 13.1, targeting the Maxeler MAX3 board (Xilinx Virtex-6 xc6vsx475t device). Each design targets a clock rate of 300MHz.

5.2 Area Results

To measure assertion costs, we add an assertion to the loop that variable `sum` is always positive (a negative number indicates overflow). We add A assertions, where $1 < \mathtt{A} < W$ by inserting the line: `if (j<A) assert(sum>0);` after the accumulation in the loop body.

Figure 5 shows area resources used (LUTs and Flip Flops) versus number of exceptions for the moving average application The cost of adding assertions lies between 5% (LUTs) and 15% (BRAMs) (relative to the area used by the original

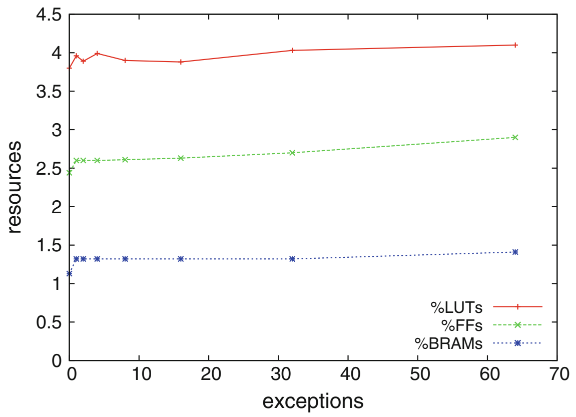


Fig. 5 Area results: % area versus no. exceptions for a 64-wide, 32-bit moving average filter

design), due to logic used to implement assertion conditions, and buffers used to store assertion results. Beyond that, there is a small linear area cost per assertion added; since each exception is a Boolean stream output, adding an exception has a small area penalty. Designers may thus add many exceptions without much concern over area costs. The exception condition has an area cost, but this could be mitigated if the toolchain eliminates any common subexpressions.

5.3 *Imprecise Exceptions*

We evaluate the savings from allowing deliberately imprecise assertions and exceptions. Figure 6 shows the results of increasing levels of imprecision in time, where zero imprecision means raised exceptions are reported accurate to the cycle they occurred on, an imprecision of one means they are reported to within one cycle, and so on. Adding just a single degree of imprecision immediately results in 5% improvement in throughput and runtime; adding more imprecision results in further improvements, but with diminishing returns; about 48% is the maximum; note that at this point the PCI express bus is almost saturated, (on this machine, the maximum is about 2200 MB/s) so little further improvement is possible. We show results for exceptions represented using both 8-bit and 16-bit output streams. It may seem wasteful to represent single-bit values with wider streams, but wider streams actually reduce runtime and increase throughput: the 16-bit is up to 12% faster than the 8-bit stream. This is due to less time spent in unpacking the exception streams on the software host.

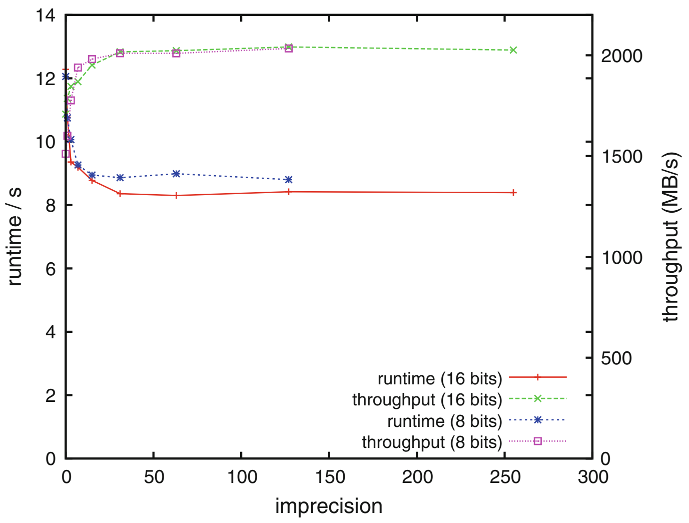


Fig. 6 Imprecise exceptions: runtime and throughput versus degree of imprecision. In this experiment, imprecision is in time, measured in cycles

6 Conclusion

We present a high-level approach for adding in-circuit assertions and exceptions to hardware designs, and a concrete implementation for Maxeler systems. Results show that our assertions and exceptions add little area and speed cost. Exception and assertion imprecision improves runtimes by up to 48%, with even a single degree of imprecision giving immediate benefits.

Current and future work includes, firstly, integrating our approach with temporal logic, allowing a more formal basis for the error handling. Secondly, add support for run-time reconfiguration. Designs could reconfigure to add exception handlers if many errors are detected, or running circuits could dynamically change exception handlers and assertions, without changing the rest of the design. Recent work has shown that unused resources in an FPGA design can be scavenged for adding monitoring circuitry, without impacting on timing or functionality [5]. Alternatively, exception handlers could be reused with a different but compatible design.

Thirdly, explore the adoption of in-circuit assertions and exceptions in enhancing the performance and capability of self-diagnosis and self-healing of critical systems, such as addressing sensor failure in avionics [14].

Acknowledgements The support of UK Engineering and Physical Sciences Research Council (EP/I012036/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, the Maxeler University Programme, Altera, Intel and Xilinx is gratefully acknowledged.

References

1. IEEE standard for property specification language (PSL): IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005), pp. 1–182 (2010)
2. Bustan, D., Korchemny, D., Seligman, E., Yang, J.: SystemVerilog assertions: past, present, and future SVA standardization experience. Des. Test Comput. IEEE **29**(2), 23–31 (2012)
3. Curreri, J., Stitt, G., George, A.D.: High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis. Int. J. Reconfigurable Comput. **2011** (2011)
4. Hoare, C.A.R.: Hints on Programming Language Design. Stanford, CA, USA, Tech. Rep. STAN-CS-73-403 (1973)
5. Hung, E., Todman, T., Luk, W.: Transparent insertion of latency-oblivious logic onto FPGAs. In: 24th International Conference on Field Programmable Logic and Applications (FPL), 2014. IEEE, pp. 1–8 (2014)
6. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. SIGSOFT Softw. Eng. Notes **31**(3), 25–37 (2006) [Online]. Available: http://doi.acm.org/10.1145/1127878.1127900
7. Scott, M.: Programming Language Pragmatics, 3rd edn. Morgan Kaufman (2009)
8. IEEE standard for floating-point arithmetic. IEEE Std 754-2008, pp. 1–58 (2008)
9. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fourth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)
10. Hung, E., Wilton, S.J.E.: Towards simulator-like observability for FPGAs: a virtual overlay network for trace-buffers. In: FPGA '13 (2013)
11. Graham, P., Nelson, B., Hutchings, B.: Instrumenting bitstreams for debugging FPGA circuits. In The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001. FCCM '01, pp. 41–50 (2001)
12. Poulos, Z., Yang, Y.-S., Anderson, J., Veneris, A., Le, B.: Leveraging reconfigurability to raise productivity in FPGA functional debug. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp. 292–295 (2012)
13. Vasudevan, S: What is assertion-based verification? SIGDA E-News, **42**(12) (2012)
14. Todman, T., Stilkerich, S., Luk, W.: Using statistical assertions to guide self-adaptive systems. Int. J. Reconfigurable Comput. **2014** (2014) [Online]. Available: http://dx.doi.org/10.1155/2014/724585
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
16. Floyd, R.W.: Assigning meaning to programs. In: Proceedings of the Symposium on Applied Maths, vol. 19. AMS, pp. 19–32 (1967)
17. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
18. Hoare, C.A.R.: Assertions: a personal perspective. Ann. Hist. Comput. IEEE **25**(2), 14–25 (2003)
19. Wang, D., Levitt, J.: Automatic assume guarantee analysis for assertion-based formal verification. In: Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific, vol. 1, pp. 561–566 (2005)

Part VII
Formal and Semi-formal Methods

From ProCoS to Space and Mental Models–A Survey of Combining Formal and Semi-formal Methods

Bettina Buth

Abstract This contribution reports of work done after the official end of the ProCoS project in 1995. Most of this work was done while the author was affiliated with Bremen University. The aim of this contribution is to show the effect of ProCoS on these projects, which comprises analysis of systems from two different application domains: space and aerospace. In both examples, the basic approach involves abstraction to CSP specifications and model-checking using FDR. Another common factor is the use of other techniques in combination with model-checking.

1 Introduction

One aim of the ProCoS project was to provide methods and tools for a systematic development of correct systems, specifically for realistic safety critical applications. This contribution summarizes projects performed after the official end of the ProCoS project in 1995. Most of this work was performed while the author was affiliated with Bremen University. These projects involve analysis of systems from two different application domains: space and aerospace. In both examples, the basic approach involves abstraction to CSP specifications and model-checking using FDR. Another common factor is the use of combinations of other techniques with model-checking.

The first example addresses analysis done for the Russian module of the ISS, especially the deadlock and livelock analysis for a realistic concurrent system of processes, which has been implemented in occam. The initial assumption that the analysis would just require an abstraction of the occam code and a simple check in FDR2 turned out to be too optimistic, since neither the abstraction nor the check could be done that easily. In the end, FDR2 was used for checking subcomponents and we used various techniques to provide a rigorous argument for the overall conjecture. Section 3 will provide an overview of this work with a focus on the abstraction and the combination of model-checking and less automatic argumentation. While much of this project has been presented on various occasions, this section also contains

B. Buth (✉)
HAW Hamburg, Hamburg, Germany
e-mail: Bettina.Buth@haw-hamburg.de

unpublished experience from a re-verification of the analysis in the context of the re-use of the software for the automatic transport vehicle (ATV).

The second experiment started at a UNU-IIST summerschool on formal methods in Beijing in 1999 during a course introducing CSP and model-checking using FDR2 [3]. John Rushby was another lecturer at this summerschool and offline a discussion about his approach analysing mode confusion situations [27] was begun. It involved a back-to-back-analysis of technical system models and mental models and was performed using the Murϕ model checker [9]. This example was intriguing because it involved two different models which were manually combined into a single model and a fairly complicated invariant that was used to capture the assumption that they are equivalent. The question arose whether the FDR2 refinement approach could be used to identify the critical deviations between mental model and actual technical system model without the added complication of the invariant. As a case study, this work was quite successful - the approach was not only suitable to detect already known problems, but it uncovered further problems in the application which had been hidden in the Murϕ model. Section 4 provides a summary of this case study - a more detailed report can be found in Buth [1], essential results have been presented at SafeComp'2004 [2].

These two examples - and several other experiments not reported here - have two things in common: the abstraction of realistic systems or parts thereof as formal models using CSP and the use of the model-checker FDR2.

2 Abstraction and Refinement–CSP and FDR2

This section provides a brief introduction to CSP and the model-checker FDR2, which were used in the verification projects to be regarded below. In addition this section contains a brief description of a selection of techniques used to combine local results or to break down a complex verification task into simpler ones. Some of these are based on CSP semantics and the relating theories, others were adapted from different analysis methods.

2.1 CSP

The specification language CSP (Communicating Sequential Processes) is associated with a formal method allowing to verify properties of systems of concurrent processes. (See Hoare [12] and the books by Roscoe [24, 26] for more details). CSP processes evolve by engaging in communications. Processes may be composed by operators which require synchronization on some communications. This, rather than assignments to shared state variables, is the fundamental means of interaction between agents. The theory of CSP is based on mathematical models of the observable behaviour of processes represented by

- *traces*: a process is represented by the set of finite sequences of communications it can perform,
- *failures*: a process is represented by its set of traces as above and also by its refusals – a set of communications it can refuse after a sequence of communications,
- *failures-divergences*, which extend the failures model with the divergences of a process – the traces during or after which the process can perform an infinite sequence of consecutive internal actions or otherwise show chaotic behaviour.

The communication behaviour of CSP is a synchronous one - two (or more) processes can engage in a communication over commona *channel* if all are available, otherwise a process is blocked on this channel. CSP provides a variety of operators for sequential as well as concurrent composition. Processes only using sequential operators can be seen as basic processes.

2.2 Refinement and FDR2

The notion of refinement is a particularly useful concept in many areas of engineering. If we can establish a relation between components of a system which captures the fact that one satisfies at least the same conditions as another, then we may replace a component by a better one without degrading the properties of the system. Refinement relations can be defined for systems described in CSP in several ways, depending on the semantic model used. The relation is noted as *Refinement* $P \sqsubseteq_X Q$. where X is `T` for trace refinement, `F` for failures refinement, `FD` for failures-divergence refinement. These refinement properties are fulfilled if the process `Q` can provide only behaviour also allowed by the process `P`.

FDR2 (*Failures Divergences Refinement*) is a model-checking tool for state machines, with foundations in the theory of concurrency based upon CSP. Its main method for establishing whether a property P holds for a CSP process system *SYS* is to investigate refinement properties: *SYS* is compared to a specification process *SPEC* for which P is known to be valid. The comparison is performed in one of the semantic models of CSP which is known to preserve property P under refinement. The main ideas behind FDR2 are presented in [23], it can be summarized as follows: Every CSP specification consisting of finite-state processes with finite-value channels can be translated into a finite *transition graph* representation. This graph contains all the semantic information of the original CSP specification. As a consequence, every property of the specification captured as refinement property can be verified by exhaustive analysis of the transition graph. In particular, the analysis of deadlock or livelock potential of concurrent systems of CSP processes can be reduced to refinement checks; in FDR2 these checks can be directly invoked. Moreover, such an analysis can be mechanized. The FDR2 tool provides this automation and has been used for all model checking results described in this document.

2.3 Abstraction

Abstraction is the basis for model-based verification of systems in general; it allows to reduce the overall complexity of a verification goal by providing a simplified perspective of the original system, which retains the properties relevant for the analysis. For the examples shown in the following different original descriptions were used as a starting point:

- occam code
- finite state models given as sets of rules or as graphical models
- architectural models of a systems describing its components and interfaces

For each of these descriptions a systematic abstraction was found to represent their essential nature as CSP specifications. These CSP specifications provided the analysis models for different investigations.

In general, a valid abstraction for a property p is required, defined as follows: A CSP specification $\mathscr{A}(P)$ is called a *valid abstraction for a property p* of the corresponding System S, if

Whenever $\mathscr{A}(S)$ has property p this implies that S has property p as well.

2.3.1 Abstraction of occam Code

If a valid abstraction $\mathscr{A}(P)$ of an occam process P is available, we only need to analyse $\mathscr{A}(P)$ instead of P. For the analysis of communication behaviour of an occam process P such as deadlock or livelock analysis the following approach in the translation from occam to CSP can be used:

1. Every sequential code sequence without communication operators is deleted.
2. Each occam channel protocol is reduced to the set of values influencing the communication behaviour in a distinctive way.
3. Every occam IF-construct `IF condition THEN P ELSE Q` may be replaced by the equivalent if-construct in CSP or by the internal choice operator of CSP yielding $P \sqcap Q$.
4. If valid abstractions $\mathscr{A}(P)$, $\mathscr{A}(Q)$ for two processes P and Q are available and these interpretations use the same abstractions on their communication interface I, then $\mathscr{A}(P) \underset{I}{\|} \mathscr{A}(Q)$ is a valid abstract interpretation of P and Q operating in parallel. Using this technique, larger abstractions can be build from abstractions on components.

Figure 1 provides an overview about the relationship between occam and CSP constructs.

occam**Construct**	CSP**Construct**
IF_THEN_ELSE	$\sqcap$ or if_then_else
PAR (PRIPAR)	$\vert\vert\vert$ or $\vert\vert$
ALT (PRIALT)	$\square$
WHILE	$P = \cdots \rightarrow P$ or P = if b then $\cdots$ else SKIP
c?x	c or c?x
c!a	c or c!a
SEQ	; or $\rightarrow$

Fig. 1 occam abstraction to CSP

2.3.2 Abstraction of Finite State Models

Automata or Finite State Machines (FSM) are an abstract formalization well understood within computer science. Variants are present in UML, design pattern and their implementation are available for almost any programming language from libraries or example implementations.

In order to verify properties of systems adhering to this specific pattern it may be of interest to transform these models to CSP models - in the following this approach will be used to investigate deviations between to automata models with regard to their observable equivalence using refinement checks.

In order to do so, it is necessary to provide a systematic transformation of an automata model to a CSP model. Table 1 provides an overview of essential par ts of an FSM and the chosen CSP construct.

This is very similar to a switch-case implementation of FSM in imperative languages.

In order to reason about state changes - as required for the equivalence check - auxiliary channels are used which report a state identification on entering the state. This allows to refine the checks further and not only compare external behaviour but also compatibility of states in a step-by-step execution. Furthermore, these auxiliary events support the fault localization should the external behaviour deviate from expected behaviour.

Table 1 Relation between FSM and CSP specifications

Element of FSM	CSP construct
State	Process
Attribute of state	Parameter of processes
Trigger events	Channel event
Conditions on transitions	Guarded commands
Multiple transitions from a state	Choice
Change of state	Call of process

2.4 Generic Theories–Pattern-Based Verification

As we know from pattern-oriented development, it is useful to identify generic classes of processes and investigate their properties without knowledge of concrete instances. This leads to generic theorems which can be exploited for the verification of instances. For the examples reported below, this has been used in two ways:

- use the generic process during abstraction directly
- derive properties of process composition based on properties of its components

In both cases it is necessary to make sure that the concrete process is a refinement of the instance of the generic theory. CSP provides generic theories of abstract processes. A generic theory provides properties valid for each process instance of the associated class.

Examples of such generic classes used in the case studies are

- `RUN(X)`
  Instances of `RUN(X)` always accept any event from set `X`. The `RUN` process can not deadlock: instances of this process class can e.g. be used for modelling environment behaviour, especially to capture specific outputs from the processes in a defined way.
- `YIELD(X)`
  For a set of events `X` `YIELD(X)` will always provide at least one element of `X`. Note that this does not exclude that one or more events are never offered, but the process will not deadlock. `YIELD(X)` was used to model the behaviour of input from the process environment.
- pipes
  consist of several processes which transfer input from their predecessor to their successor in a serial way. If the basic processes of the pipe do not deadlock, then the overall process will not deadlock, thus it suffices to verify deadlock freedom of the basic processes
- buffers
  store data up to a certain capacity and allow to retrieve these data in *first in-first out* order. Properties of buffers are well known.

2.5 Algebraic Reasoning

CSP possesses a rich proof theory which provides algebraic laws for the transformation of CSP processes into equivalent ones. Specifically, deadlock and livelock freedom are preserved under application of these laws. Algebraic manipulation is useful to simplify process terms and reduce the state spaces of processes before starting the model checking process.

2.6 Compositional Proof Theory

The strongest way of exploiting compositionality is by using theorems which allow to reduce the overall verification obligation to obligations for individualing components of the system. A simple example is the following:

Let $SYS = P \,|||\, Q$ be a system of two processes P and Q which run concurrently without any synchronization. Then SYS is free of deadlocks whenever P and Q are free of deadlock individually.

Another way of exploiting compositionality is to refer to an essential property of refinement in connection with operators for the composition of processes (preservation of refinement):

If $P_i \sqsubseteq_{FD} Q_i$ for $i : 0..n-1$ and ω is an n-ary operator,
then $\omega(P_0, \dots, P_{n-1}) \sqsubseteq_{FD} \omega(Q_0, \dots, Q_{n-1})$ holds.

In connection with generic processes, compositionality can be used to reduce the overall verification task so simpler ones.

3 Space

The acceptance of Formal Methods in industries essentially depends on their scalability, i.e. their applicability in large scale realistic industrial projects. An important aspect is the availability of suitable tools, but from our experience this is but one aspect. The diverse nature of system components and the techniques used in the different steps of the development process require the use of a combination of methods for the development as well as for the analysis of these components. In this section we report experiences using a combination of methods for the analysis of a large software system, namely the fault-tolerant data management system for the International Space Station (ISS).

Deadlocks and livelocks are two essential problem areas for systems consisting of concurrent processes. This section presents a summary of experiences gained during the analysis of a fault-tolerant computer for the Russian module of the international space station ISS between 1995 and 2004. The original analysis of the relevant software was performed at the end of the design phase and a re-verification during the coding phase. Deadlock and Livelock Analysis are part of a larger verification suite, which validates different aspects of the overall system. They employ a combination of techniques based on local results gained from model-checking abstractions of software processes using CSP and FDR2.

Various aspects of the original work have been published; see e.g.

- deadlock analysis: Buth et al. [5],
- livelock analysis: Buth et al. [6, 7]

- detailed description of abstraction methods and verification of the Byzantine agreement protocol: Peleska et al. [21],
- load analysis using GSPNs: Schlingloff [32], and
- efficient use of generic theories in the verification of fault-tolerant systems: Buth et al. [4]. More detailed description of some of the aspects can be found in Buth [1], another overview in Peleska [20].

Some of this material has been reused in this survey.

After a brief introduction to the system under analysis, this section provides an overview of the techniques employed for the analysis of deadlock and livelock freedom of the system components.

3.1 Technical Background: The Fault Tolerant Computer

The software analysed is part of a fault tolerant computer that is used in the International Space Station (ISS) to control space station assembly, reboost operations for flight control, and data management for experiments carried out in the space station. In the following, the system architecture and the goal of the analysis as well as the starting point for the verification are described.

3.1.1 FTC Architecture

The overall architecture consists of up to four communicating lanes, each providing services for the applications. Each of these lanes is structured into an application services layer (ASS), a fault management layer (FML), and the avionics interface (AVI). The ASS resides on the application layer board and contains table driven services for the application software and the operating system. The AVI is in charge of the MIL Bus protocol handling according to predefined timing slot allocations. These are defined in an input/output table. The function of FML is twofold: First, it provides the interface between ASS and AVI of one lane, transferring messages from AVI to ASS and vice versa. Second, it performs the data transfer between lanes thus allowing communication between the fault management layers of all lanes. This communication is the basis for error detection, error correction, lane isolation (in the case of an unrecoverable error), and lane reintegration. In each lane, the application layer plus ASS runs on a customized Matra board using a SPARC CPU. Both FML and AVI reside on separate transputer boards. The lanes communicate only at FML level using the transputer links. Each FML uses up to three links for communication with the other lanes, and one link (link 0) for communication with AVI. Data transfer with ASS is performed using a VME interface. See Fig. 2 for the architecture of a full four-lane system.

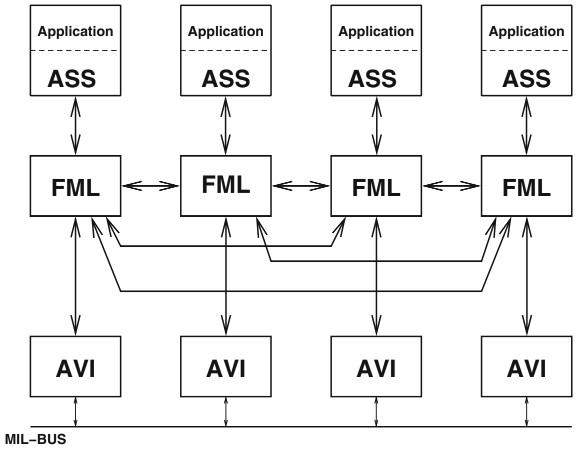


Fig. 2 FTC architecture

Error detection is essentially based on a two round Byzantine distribution schema [14] where data is communicated between FMLs and voted using various specialized voters. The aim is to ensure that

- all ASS instances of non-faulty lanes get identical messages from FML,
- all AVI instances of non-faulty lanes get identical messages from FML,
- for data calculated by all lanes (so-called *congruent source messages*) all non-faulty lanes get the correct(ed) message,
- for data calculated by one lane (single source messages) all non-faulty lanes get the correct(ed) message if the originator is not faulty.

The implemented design allows detection of one Byzantine or deterministic fault in a four-lane system and recognition of a deterministic fault in a three-lane system.

3.1.2 Goals of Communication Behaviour Analysis

Both FML and AVI software are implemented in occam [13], consisting of systems of processes running concurrently and communicating via internal channels and partly using shared memory to exchange data without the additional communication overhead. The software is structured hierarchically: larger processes, each consisting of several subprocesses, implement different parts of the respective functionality and communicate the computation results to the other main processes. In addition to the usual aspects of functional correctness and timing, a concurrent architecture of software adds the new problem of potential blocking. It must be guaranteed that the software does provide its services to the adjacent components of the overall system. Blocking can be due to two causes: either the system does not work at all because components mutually wait for results, or it does only work internally without communicating with its environment. The first situation is called a *deadlock*, the second a *livelock*. Our essential task for the analysis of the communication behaviour was to show that neither deadlock nor livelock situations can occur in the implementation of AVI and FML.

For the deadlock analysis we have to check whether any of the communicating system of processes that forms the software can continuously block the communication with the environment. Formally, the following verification goal had to be investigated:

> In an environment that always accepts outputs from the system but may or may not refuse to provide inputs, the following assertion holds: Whenever the system reaches a stable state where all internal communications are blocked, the system will always accept new input from the environment.

In complex hierarchically organized systems of concurrent processes, communication between low-level sub-components is usually concealed from the environment. occam provides suitable operators to implement such hiding of internal communication. Situations, where the interaction of internal components result in uninterrupted chatter between these components without visible progress at the interface channels are known as *livelock*.

The objective of our livelock analysis is to investigate occurrence of internal divergence only.

> A CSP (or occam) system X is called *livelock free* with respect to interface channels $c_1, c_2, \ldots, c_n$, if the system will never engage in an unbounded sequence of (internal) communications without allowing (visible) communication on the (external) interface $c_1, c_2, \ldots, c_n$.

This property is as important as the *absence of deadlocks* since it ensures the interaction of the system with its environment.

3.2 Verification Approach

The material provided as input for the analysis consisted of

- a printed version of pseudo code for AVI,
- source code of the final implementation of FML and AVI,
- diagrams for the overall architecture and communication behaviour of individual processes,
- general information about the system requirements and technical details of the system,
- verbal communication with the system engineers.

The general idea for the analysis of communication properties of occam programs as used in the project is to exploit that CSP is an abstract specification for occam code. Thus for both deadlock and livelock analysis performed the occam code is abstracted to CSP and the model-checker FDR2 [10] is used to evaluate the relevant properties for the model. Provided the CSP model ia a valid abstraction, the check on the model allows to deduce properties of the code. An early experience: after manually abstracting the occam programs to CSP processes, the system still is too large for a direct approach using FDR2. Thus it is necessary to decompose the task and use several of the other techniques mentioned above for combining the results to obtain an overall result for the full system.

The mathematical proof theory of CSP allows to verify properties of CSP specifications by means of formal reasoning. The CSP language, its mathematical foundations and its possible applications have been thoroughly investigate since the late sixties (see [8, 12, 25, 26, 31]). For the verification goal described above we apply various verification techniques. They can roughly be divided into those which are applied to basic components and those which are used for combining the respective results on the basic components

The techniques in the first group are

- *Abstraction* is applied to "lift" occam process components to CSP components reflecting the essential aspects of the process communication behaviour while abstracting from details irrelevant for the verification goal,
- *Generic theories* increase the efficiency of analysis: process instances of a generic class inherit the class properties, which means that it is only necessary to show that concrete processes are instances of a suitable generic process,
- *Algebraic reasoning* is applied to transform CSP process specifications into equivalent ones better suited for the model checker.
- *Model checking* is used for the mechanized verification of small-sized CSP components for both deadlock and livelock freedom.

Those in the second group are:

- The *compositional proof theory* of CSP is applied in connection with refinement properties to derive global properties of the complete system from the local properties established for the isolated components,
- *Liveness induction* provides a suitable iterative approach extending the system step-by-step and making use of local livelock freedom results of the component processes [7]
- A form of *cycle analysis* is used on the data flow diagrams to show that cycles in the communication can not induce an infinite backlog as cause of a livelock [5],
- A *dependency analysis* is applied in order to reduce the number of cycles to be investigated for livelock analysis [1, 7]

The size of the FTC system provided an obstacle for the fully formal verification of the system with regard to deadlock and livelock. By decomposing the overall verification task into smaller subtasks we were able to provide the analysis required based on a combination of formal reasoning, model-checking and rigorous arguments.

Table 2 Goals of FTC analysis

Verification goal	Verification method
Deadlock freedom	CSP: generic theories - abstraction - model checking - algebraic laws - compositional theory - cycle analysis - liveness induction
Livelock freedom	As for deadlock analysis plus dependency analysis
Absence of bottlenecks	Stochastic petri nets
Correct implementation of Byzantine Protocol and failure detection	Abstraction - model checking - compositional proof theory
Correct implementation of application services	Verification using Hoare Logic
"Most non-deterministic admissible software/hardware integration is correct"	Hardware-in-the-loop-testing

Table 2 gives an overview of the methods used for the communication behaviour analysis as well as the other verification and test goals for the FTC system.

3.3 *Lessons Learned–Part 1*

This subsection summarizes the experiences made during the deadlock and livelock analysis for the FTC. These relate to deriving valid models suitable for model-checking, the combination of formal verification and rigorous arguments, the scalability of the model-checking tool FDR2 and the reuse of models and results for re-verification.

3.3.1 Abstraction of occam Code and Interfaces

Since deadlock and livelock analysis are essentially properties of communication behaviour of processes in a system of concurrent processes, the models can abstract from large parts of the sequential parts of the code. The models thus represent a kind of communication skeleton of the code, sequences of communication events determined by internal control flow of each process. Similarly a first model can abstract from data communicated, thus reducing the number of possible events in the CSP model. The derivation of these models can be done in a systematic way - could possibly even be automated.

This reduction to causal dependencies between communication events introduces a high level of non-determinism to the models of processes; not all of the sequences represent realistic communication sequences - but the realistic sequences are a subset. With regard to the validity of the results, the approach is still sound - if the larger set of communication sequences does not allow deadlocks, the subset does neither. On the other hand, false positives are potentially introduced and the overall state space for model checking grows. In order to reduce this effect, the initial models were refined by introducing details of the code into the model, such as specific conditions guarding individual communications or details of the date communicated via channels.

Introducing details can in some cases also increase the state space to be investigated by the model checker. If the state space becomes too large, it may be necessary to decompose the checks.

The overall model-derivation is thus an iterative process, which requires insight into the application as well as experience with the constraints of FDR2.

3.3.2 Combining Methods

As explained in the preceding sections, the verification steps involved a number of semi-formal arguments that were carefully reviewed but could not be checked in a mechanical way. Therefore it is necessary to assess the completeness of the verification results obtained and the test coverage achieved. Due to the usage of very simple communication patterns in the FML software design our confidence into the adequacy of the abstractions and the completeness of the verification is very high for the FML software. For technical reasons, the AVI layer could not be designed with such simple patterns. As a consequence, the deadlock and livelock analysis results obtained for the AVI should rather be regarded as a "sophisticated test suite" which after having uncovered a number of errors did not find any new ones. However, it should be emphasised that the quality of these "tests" is much higher than what could ever be achieved by systematic but informal design reviews because of the high number of states explored during the model-checking process.

The verification effort was much smaller for those parts of the system where standardised design patterns had been used by the software developers. Optimization for run-time efficiency introducing short-cuts for exceptional situations proved to be one of the main obstacles for adhering to such pattern.

3.3.3 Differences Between Deadlock and Livelock Analysis

Although deadlock and livelock verification both start from the same code, it is not directly possible to use the same abstractions. Due to the hierarchical architecture of both components the freedom of livelock for each of the main processes was tried to be established first. During this phase some adjustments in the abstractions were necessary to eliminate non-determinism and formalise the new proof obligations. One example is the explicit introduction of timer processes in order to avoid divergence introduced through events depending on specific timers.

During these first step of livelock analysis it became obvious that it would not be feasible to use model-checking directly for the subprocesses of FML and AVI even in cases where it was possible for the deadlock analysis. On the one hand this is due to the changes in the abstraction which enlarge the state space, on the other hand the problem arises since the states themselves are larger. The reason for this is that livelock analysis uses the *failure-divergence* model of CSP, while the deadlock analysis could be performed within the *failures* model. The internal representation of the states has to contain the additional information about the divergence sets and thus is larger.

Similarly, the effort for combining local results of subprocesses required a more creative approach using additional techniques. Two different approaches were employed in this situation:

- further abstraction and exploitation of the theorems for preserving the results under refinement,
- further decomposition of components, separate analysis for each basic unit, and derivation of properties for the combined units.

While the first proved to be a suitable way of dealing with the main processes of FML the complex communication behaviour of AVI made it necessary to pursue the second approach. In both cases it was necessary to employ suitable means for combining the results for the overall unit, such as liveness induction and dependency analysis.

Last but not least it should be mentioned, that the overall project was a successful one. The formal analysis uncovered a number of critical scenarios for the application of the FTC system, which could not be found be conventional tests. The overall verification suite provided essential input for re-engineering certain parts of the system and these again were analysed in a re-verification effort.

3.3.4 Reuse of Models and Verification Results

Several years after the original verification project, the occam code developed for the FTC was reused in the ATV–the Automatic Transport Vehicle transporting goods to the ISS. The code was fromally re-used, but according to ECSS standards up to 20% of the code can be modified and still from development perspective it is a reuse.

The open question was whether the changes to the code invalidates the results of the deadlock and livelock analysis. To check this, first an evaluation of the changes was performed - again with a focus on communiation order and channel usage; the latter referring to changes in the channel protocol as well as to the concrete data used in each communication event. In a second step the effect of these changes to the model were investigate. The following table summarizes the effect of typical classes of changes to the model.

Change	Potential Impact on Model and Results
Renaming of channels	Does not invalidate the analysis result (renaming provides equivalent processes in the model)
Protocol changes	Can effect the model unless the details of the protocol are omitted in the model; otherwise similar effect as for removal and or new introduction of channel
Removal of channels	Can effect both deadlock and livelock behaviour, both locally and globally; for local results the situation may be uncritical if another case with the same behaviour exists (these can be identified in a more abstract model) - requires more detailed analysis to determine specific situation
New channel or communication events	Can effect both deadlock and livelock behaviour, both locally and globally; for local results the situation may be uncritical if another case with the same behaviour exists (these can be identified in a more abstract model) - requires more detailed analysis to determine specific situation
Re-ordering of cases (ALT or IF)	Without effect on the model
Removal or introduction of cases	Similar to removal or introduction of channel or channel events
Splitting of a process into two parallel processes	Will in general effect the model and the validity of the results

As a consequence of these finding, an adaptation of the models according to the changes was made and the analysis re-run. As it turned out, the changes did not introduce new deadlock situations, but a new livelock situation was found. Further analysis identified an older problem, which had been waved as non-realistic in the original verification project.

The overall experience from this follow-up project is that the classification of changes provides a suitable basis for a systematic adaptation of the models and that the re-run of the analysis in FDR2 was without any problems.

4 Mental Models and Refinement

The ever-increasing complexity of computer-based systems has lead to a changed role of human operators, especially in safety-critical applications such as air-craft and train control, chemical and nuclear plants, medical equipment, or automobile components. The use of computers in such systems has a high potential for automation as well as extended functionality, but also requires sophisticated control and monitoring mechanisms due to the inherent complexity. These themselves can be implemented as computer-based processes which allow to take away the strain from human operators who would otherwise have to cope with a multitude of information and a higher demand on reaction times required for the interaction with such systems.

Nonetheless, in many applications a total automation of the system control is not accepted or (not yet) possible. Human operators are often the ultimate instance for dealing with emergencies or have to react to information not directly available to the computer-based kernel systems. Research activities in the Human Factors community focus on human-computer interfaces based on psychological as well as design-oriented considerations. As Sarter et al. [29] and Leveson et al. [16] point out, technology-centred automation potentially leads to designs which are problematic for the human interaction.

One area which has found attention is the investigation of so called *automation surprises*, particularly *mode confusion*. This section discusses the use of model-checking for the comparison of abstract system models and mental models with the objective to analyse mode confusion situations. The emphasis is on the CSP specification and model-checking aspects rather than on the socio-technological perspective. The remainder of the introduction provides the background for the approach as well as an informal description of the example, a *kill-the-capture* scenario. Section 4.3 describes one possible approach to the analysis of this example using FDR2. In contrast to other attempts using model-checkers for this task, the refinement approach allows a direct comparison of the two models which can be easily derived from a rule-based description or an finite automata model. Section 4.4 summarizes the experiences and tries to generalize the results.

4.1 Mode Confusion Analysis–Background

Modes are identifiable and distinguishable states of a system which differ with regard to the effect of interactions. The complexity of a system is reflected in the number of different modes and complex rules for mode transitions as well as functionality in a mode. Mode confusion scenarios – also called automation surprises or feature interaction depending on the application domain – describe situations where the operators assumption about the system mode differs from the actual mode of the system and actions performed under this assumption result in critical situations. In order to detect and eliminate mode confusion, a thorough analysis of the system design and functionality as well as the human-computer interface is required.

Techniques from the formal methods field prove to be useful for mode confusion analysis. Several approaches based on abstract models of the system are documented; see e.g. Leveson et al. [16], Miller and Potts [18], or Lüttgen and Carreño [17]. These experiments focus on the identification of situations that potentially lead to mode confusion. Rushby [27, 28] suggests a complementary use of model-checking based on two different models of the system. The *actual model* is an abstract model of the actual system behaviour; the second called *mental model* reflects the view of the operator which may be a reduced version of the full model or even may contain wrong assumptions about the system.

In contrast to the approaches of Leveson and Miller and Potts, Rushby's approach aims at identifying critical discrepancies between the models rather than investigating the mode confusion potential of the actual model. Rushby formalizes the models in the Murϕ [9] model-checker notation and employs Murϕ to perform a full state exploration. Since Murϕ is not able to compare two models directly, both models are merged into one by renaming the relevant state components such that these have disjoint names. The Murϕ rules then are used to describe the effect of inputs or events to the full set of state variables. An invariant is employed to specify that both models are in equivalent states after each step.

This is slightly unsatisfactory, since an untrained person will not be able to determine such a specification from the distinct views of the actual and mental models respectively, even if the Murϕ rules can be easily understood with a basic knowledge of state transition machines or simple automata. Similarly, the formalization of invariants as criteria for the absence of mode confusion will in general require some explanation or even a manual analysis of the models (which may very well uncover the problems in the models). Rushby himself [27, 28] suggests to employ a different type of model-checker, namely the CSP-based tool FDR2 as an alternative, since FDR2 allows to compare models in a more direct way. In the following, this suggested approach is investigated, taking the Murϕ model as a starting point.

4.2 The Example

The example in Rushby's papers [22, 27, 28] is taken from an article by Palmer [19], which reports two cases of altitude deviation scenarios. These cases and three others were observed in a NASA study in which several crews flew realistic missions in DC-9 and MD-88 aircraft simulators. This example has previously been investigated by Leveson [15].

In the following, the scenario description as stated by Rushby [27, 28] is presented, which is the starting point for the Murϕ model. In order to follow the scenario it is necessary to explain some features beforehand.

The system behaviour depends on two modes: The `PITCH` mode is a control element for the autopilot which determines the climbing behaviour of the aircraft. The modes are

`VERT SPD`	vertical speed; climb at a specified rate (feet per minute)
`IAS`	indicated air speed; climb at a rate which is consistent with holding the air speed (knots)
`ALT HLD`	altitude hold; hold current altitude
`ALT CAP`	altitude capture; provide smooth levelling off when reaching desired altitude

The second relevant aspect is the `ALT` capture mode (one of several possible capture modes) indicates that the aircraft should climb to the the desired altitude and then hold that altitude. For the example it suffices to imagine this mode as a binary value which reflects whether the mode is set (armed) or not.

The interaction between the modes is of particular interest:

- if ALT capture is armed and the desired altitude is reached, the pitch mode is set to ALT HLD.
- the ALT CAP pitch mode is entered automatically when the aircraft gets near the desired altitude under the condition that ALT is armed; it switches off the ALT capture mode
- if ALT CAP pitch mode is set and the desired altitude is reached, the aircraft levels off and pitch mode is changed to ALT HLD.

The scenario as reported by Palmer [19] describes a potentially critical situation where an aircraft leaves its assigned flight corridor and enters a flight altitude which could be assigned to other aircrafts. The cause for this situation is obviously that the ALT capture was switched off without the Captain noticing it (the only information provided is that one of a large number of flags switches to blank). Analysis of the situation shows that the interaction between pitch modes and ALT capture mode is more complex than first assumed.

Rushby [22] also derives a state machine representation of the abstract behaviour of the autopilot with regard to pitch mode and altitude capture mode. This model takes into account the relevant modes and the inputs of both the plane crew and the events from the environment. This model, which is shown in Fig. 3 abstracts from

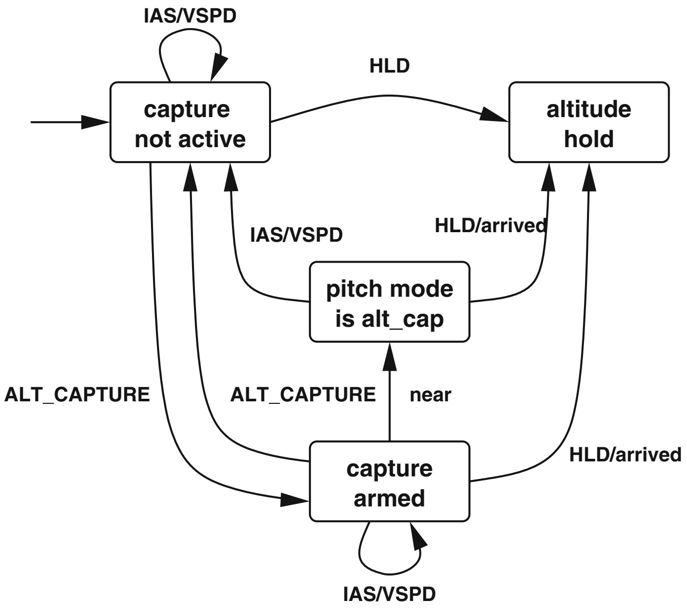


Fig. 3 State machine for actual model

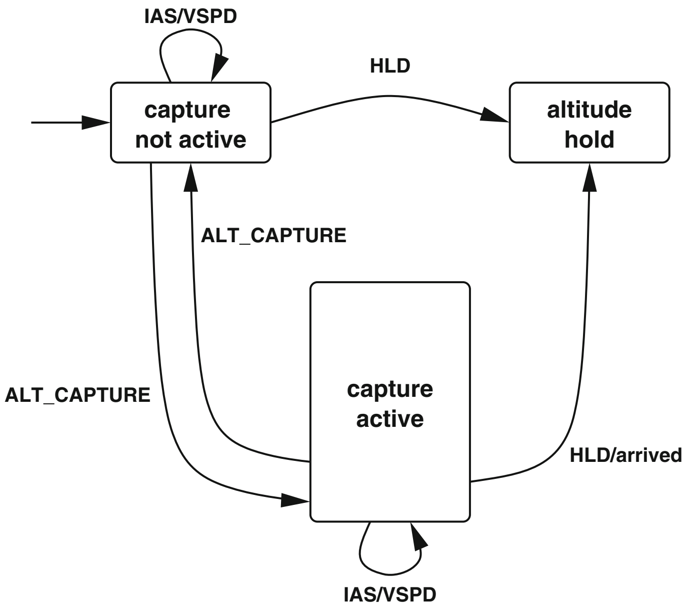


Fig. 4 State machine for mental model

the general status of the plane, as for example altitude, speed, motion or similar and from related values the pilot could enter. What remains is an abstraction of the behaviour focused on pitch mode and capture mode restricted to ALT. Similarly, Rushby provides a state machine representation of the mental model as derived from the case study. This is shown in Fig. 4.

The obvious difference between the two models is the number of states. The mental model does not contain an explicit state for ALT CAP, the pitch mode which is entered automatically without pilot interaction. This omission models the fact that the pilot was not aware of this particular mode and the related changes to the ALT capture mode. A formal analysis of these automata models with regard to their language equivalence will also reveal deviations in the possible mode transitions, but further analysis is required to examine whether these differences are indeed critical.

4.3 *Verification Approach*

This section presents an approach of employing FDR2 [10] for the investigation of the mode confusion situation reported by Palmer [19]. The Murϕ specification described by Rushby provided the starting point. The central question is whether the models can be presented in a user-friendly form as CSP specifications and whether the FDR2 refinement checks provide an adequate means for identifying critical deviations relating to mode confusion.

For this purpose several specifications of the models were investigated, which involved both rule-based and automata-based models, where the latter is based on the automata for actual and mental model as presented in Figs. 3 and 4. For the experiment the error situations uncovered by FDR2 were compared to the problems found using the Murϕ system. A full report of this and the other approaches can be found in Buth [1], parts of this has also been published in 2004 [2].

This subsection summarizes the possibilities of checking mental and actual model by comparing them using the refinement properties of CSP models.

The modes are modelled as enumeration types in CSP_M transition events as channels. In order to be able to observe the change of states additional artificial channels are introduced, which report entry of a state and the respective transition event.

Two distinct processes `ASYS` and `MSYS` are specified for the actual and the mental model, respectively. The state spaces of both models are determined by the pitch-mode and the capture-mode, which are stated as parameters of the processes.

The specification of the mental model can easily bee derived from a set of rules or an automata representation such as the one in Fig. 4. In particular, it can be derived without information about the actual model. Note that for the analysis of the mode-confusion situation such a rule-based or automata model will originally need to be derived from the understanding of the operator rather than other material. For this study, the mental model was taken from the Murϕ example.

In order to prove that the models are equivalent, it is necessary to map them to a common set of observable events.

Now it is possible to specify the desired equivalence. Since FDR2 does not provide a direct means for checking equivalence, it is necessary to check mutual refinement of the processes `ASYS` and `MSYS`. For mode confusion analysis it is of interest to prove equivalence both in the trace as well as the failures model. Trace refinement (`[T=`) in FDR2, only ensures that both systems are able to perform the same sequences of events. For the given example this ensures that both systems are able to react to external inputs in the same way and that the state changes are performed accordingly. In addition, it is of interest to know whether at any point in time one of the models could refuse an event which can not be refused by the other. Refusal properties are checked using failures refinement (`[F=`) in FDR2.

The checks reveal that the actual model is not a trace refinement and thus neither a failures refinement of the mental model. Analysis of the error scenario uncover that it reports the known problem detected by Murϕ. A detailed analysis of the errors reported can be found in [1] or [2].

After performing the corrections analogously to the suggestions for the Murϕ model, the resulting specification still shows an error.

Checking the traces and refusals reveals that while the actual model is not allowed to perform `ALT_CAPTURE` or `near`, the mental model could also engage in `ALT_CAPTURE`. Further analysis of this situation in comparison with the error-free Murϕ model reveals a flaw in that model: rule `"ALT CAPTURE"` in the Murϕ specification reads as follows:

```
rule "ALT CAPTURE" pitch_mode != alt_cap ==>
begin
  capture_armed := !capture_armed;
  ideal_capture := !ideal_capture;
end;
```

This means that the behaviour of the mental model, namely the changes to state variable `ideal_capture` are influenced by the value of `pitch_mode`, which is not part of the mental model - a problem introduced be merging the two models. The FDR2 error shows the effect of the change to the actual model alone and reveals a new error situation. This error situation is due to the change with regard to the error found above: guarding `ALT_CAPTURE` in the actual model prevents a second such event in pitch-mode `alt_cap`, but in the mental model such a change is allowed. Thus the corrections still do not capture the problems arising from the hidden state properly.

4.4 Lessons Learned–Part 2

This section summarizes and generalizes the experiences using FDR2 the suitability of model-checking for mode-confusion analysis in general and the exploitation of mode-confusion analysis for system design.

4.4.1 Evaluating the FDR2 Approach

The essential difference between the Murϕ and the CSP$_M$ specification is the way in which the models are compared. The FDR2 specifications allow a separate specification of actual and mental model, while the Murϕ specification presents a view of the combined models with a partially shared state space.

For both approaches it is necessary to determine how mode confusion situations can be identified, which parts of the specifications need to be monitored to detect such a critical situation. The study presented in Buth [1] contains three different variants for the example: the first directly corresponding to the Murϕ version, a second separating the actual and mental model but still using the rule-based description as basis, and a third directly derived from the automata representation as given in Figs. 3 and 4.

The overall experience with modelling these versions in FDR2 is quite encouraging: each of the models requires little effort for a CSP expert or even someone with a general specification background. Similarly, the evaluation of error scenarios reported by FDR2 does not pose any particular obstacles assuming a basic understanding of the overall system functionality.

As the FDR2 approach using separate models allowed to reveal additional errors, the benefit is obvious: it avoids the more complex combination of state spaces and definition of invariants which is required for using Murϕ.

Whether the models are derived from rules or from automata depends on the original data provided; in either case the specification can be generated in a systematic way, which probably could also be automated.

4.4.2 Model-Checking for Mode Confusion Analysis

A first conclusion from the experiments with FDR2 is the confirmation of Rushbys résumé: model-checking provides a relatively easy approach to investigating models with regard to mode confusion situations. One particular benefit is the ease with which the models can be adapted and extended in order to check potential corrections. At least with the given example the model-checker provides almost immediate feedback on error situations.

Two essential questions need to be discussed with regard to the general usage of model-checking for this kind of task:

- How can the specifications for mental and actual model be derived in a systematic way and on basis of which input?
- How can the errors found by model-checking be related to situations in the real system?

Both questions are strongly connected to the topic of suitable abstraction for both the real system and the operators understanding of the system. With regard to the application of a model checking tool, the specifications should be as abstract as possible, restricted to the minimal set of state and environment information. This is a prerequisite for a successful application of a model-checking tool since too much information will in general lead to a state explosion and thus to potential problems with the state-exploration approach.

A discussion of the questions concerning abstraction and error analysis in relation to the adequacy of the models for the presented example can be found in Buth [1]. The general conclusion is that the suitability of the abstraction and form of specification depends on the concrete application and the knowledge of the people involved; a systematic approach will only be possible when more experiences with this use of formal methods in the framework of human-computer interfaces are available.

Although the results presented in this study are quite encouraging and point to a very interesting direction of using model-checking and formal methods in general, some remarks are due concerning the applicability of this approach. The example considered here as well as those discussed by the other authors, are fairly small parts of larger and more complex systems. It requires more examples to prove that the approach scales to realistic applications.

5 Conclusion

This section summarizes the overall experience gained from the projects in view of the use of formal methods and CSP refinement. In addition to the above one other project is briefly summarized which also used refinement proofs employing FDR2: the verification of a fault-tolerant communication system. This example is more directly related to a step-wise development justified by refinement proofs as suggested by ProCoS.

5.1 Overall Evaluation

While the experience reported in this chapter are only exemplary, they nonetheless provide encouragement for the use of formal methods, specifically CSP and model-checking, for some verification tasks of realistic systems. The recommendation is not to replace other forms of analysis such as static analysis or tests of the respective systems, but to identify specific properties and criticality levels for which the additional effort of this type of analysis is justified. In such cases a promising approach is a joined team of experts in the application domain and experts in formal methods and tools. Tool support will be essential to manage the number of tasks related to such a verification project.

Abstraction provides one possible approach to focus on specific properties, but the systematic derivation of models will still be a significant part of the work to be done and requires experience and a certain level of creativity. Since this part strongly depend on the basis material (code, other models, informal information) the potential for automation can only be evaluated for certain settings - in general a more formal input will increase the chances. A strong semantic connection as the one between occam and CSP provides a suitable starting point.

FDR2 proved a very suitable tool for both verification efforts, at least to establish local results on subprocesses. Refinement was used for different purposes:

- in the FTC example refinement allows to reduce the individual verification tasks and semantically deadlock and livelock analysis are special variants of refinement.
- in the mode-confusion analysis mutual refinement establishes equivalence.

5.2 Other Experiments

Refinement has been an integral part of the ProCoS approach for developing safety-critical systems. During the research visit of Michael Schrnen from Capetown at Bremen University, we had the opportunity to use this idea for the justification of his architecture for a fault-tolerant train communication system. By using increasingly more detailed abstract models of his system, it was possible to ensure that the essential requirement of reliable communication was indead achieved through the redundancy mechanisms used in the system.

The project started as a research activity at the University of Cape Town with the objective to investigate the possibilities to replace the original parallel transmission of data between interlockings by a serial transmission system which could use a variety of serial channels. In addition, the serial channels had to be transparent to the existing interlockings. This was achieved through the development of fail-safe data transceivers which were connected to each interlocking (see Schrönen [30] for details).

In order to justify that the architectural design of the fail-safe communication system is sound it was necessary to check that the original requirements for the communication are still fullfilled by the replacement system. The approach involves a stepwise refinement of the system and its subcomponents including the verification that each refined view is a valid implementation of the next abstract level. With increasing detail it is required to reduce the verification tasks to simpler ones and justifying the overall result using compositionality with regard to refinement. Details of this example can be found in [1].

Since the projects reported here have been concluded, FDR2 has been developed further - since 2013 a new version FDR23 [11, 33] is available, which provides a modern user interface as well as a multi-core architecture to improve scalability. FDR23 also integrates several other tools developed in the context of CSP such as a step-by-step interpreter of CSP specifications called probe. Whether the new system could also improve on the original results with respect to proving larger processes has not been investigated. Currently it is used in a student project investigating the use of CSP refinement for analysing the potential for safety-related failure modes at HAW Hamburg. The idea is similar to the one for mode-confusion, but yet only a concept has been developed.

References

1. Buth, B.: Formal and Semi-Formal Methods for the Analysis of Industrial Control Systems. Bremen University (2002)
2. Buth, B.: Analysing mode confusion: an approach using FDR2. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) Computer Safety, Reliability, and Security. Lecture Notes in Computer Science, vol. 3219, pp. 101–114. Springer, Heidelberg (2004)
3. Buth, B., Peleska, J.: Formal methods for large-scale industrial applications – deadlock and livelock analysis for the international space station. In: Tutorial Material for the Advanced Summer School in Formal Methods and Applications, Beijing, China, October 1999
4. Buth, B., Cardell-Oliver, R., Peleska, J.: Combining tools for the verification of fault-tolerant systems. In: Berghammer, R., Buth, B., Peleska, J. (eds.) Tools for Software Development and Verification. BISS Monographs, vol. 1. Shaker-Verlag (1996) (in print)
5. Buth, B., Kouvaras, M., Peleska, J., Shi, H.: Deadlock analysis for a fault-tolerant system. In: Johnson, M. (ed.) Algebraic Methodology and Software Technology. Proceedings of AMAST'97. LNCS, vol. 1349 , pp. 60–75. Springer, December 1997

6. Buth, B., Peleska, J., Shi, H.: Combining methods for the analysis of a fault-tolerant system. In: Haeberer, A.M. (ed.) Algebraic Methodology and Software Technology, Proceedings of AMAST'98. LNCS, vol. 1548, pp. 124–139. Springer, January 1999
7. Buth, B., Peleska, J., Shi, H.: Combining methods for the analysis of a fault-tolerant system. In: Proceedings of Quality Week '99, May 1999. (CDrom)
8. Davies, J.: Specification and Proof in Real-Time CSP. Cambridge University Press, New York (1993)
9. Dill, D.: The murϕ verification system. In: Alur, R., Henzinger, T. (eds.) Computer Aided Verification, CAV'96. LNCS, vol. 1102. Springer, Heidelberg (1996)
10. Formal Systems (Europe) Lts. FDR2 User Manual, FDR 2.97 edition. http://www.fsel.com/documentation/fdr2/html/fdr2manual.html
11. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — a modern refinement checker for CSP. Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 8413, pp. 187–201. Springer, Heidelberg (2014)
12. Hoare, C.A.R.: Communicating Sequential Processes. Red Series. Prentice-Hall International, Englewood Cliffs (1985)
13. inmos ltd. occam 2 Reference Manual. Series in Computer Science. Prentice Hall International (1988)
14. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
15. Leveson, N.G., Palmer, E.: Designing automation to reduce operator errors. In: Proceedings of the IEEE Systems, Man, and Cybernetics Conference (1997)
16. Levevson, N.G., Pinnel, L.D., Sandys, S.D., Koga, S., Rees, J.D.: Analyzing software specifications for mode confusion potential. In: Johnson, C.W. (ed.) Proceedings of a Workshop on Human Error and System Development, Glasgow, Scotland, Glasgow Accident Analysis Group, Technical Report GAAG-TR-97-2, pp. 132–146, March 1997
17. Lüttgen, G., Carreño, V.: Analyzing mode confusion via model checking. Technical Report NASA/CR-1999-209332, ICASE Report No. 99-18, ICASE - NASA Langley Research Center, May 1999. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.5171&rep=rep1&type=pdf
18. Miller, S.P., Potts, J.N.: Detecting mode confusion through formal modeling and analysis. Technical Report NASA/CR-1999-208971, NASA Langley Research Center, January 1999. https://shemesh.larc.nasa.gov/fm/papers/Miller-99-cr208971-Mode-Confusion.pdf
19. Palmer, E.: Oops, it didn't arm. A case study of two automation surprises. In: Jensen, R.S., Rakovan, L.A. (eds.) Proceedings of the 8th International Symposium on Aviation Psychology, Columbus, OH, The Aviation Psychology Department of Aerospace Engineering, Ohio State University, pp. 227–232, April 1995.
20. Peleska, J., Buth, B.: Formal methods for the international space station iss. In: Olderog, E.R., Steffen, B. (eds.) Correct System Design - Recent Insights and Advances. Lecture Notes in Computer Science, vol. 1710, pp. 363–389. Springer, Heidelberg (1999)
21. Peleska, J., Shi, H., Kouvaras, M.: Combining methods for the analysis of a fault-tolerant system. In: Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing (PRDC 1999) (1999) (Submitted)
22. Rushby, J., Crow, J., Palmer, E.: An automated method to detect potential mode confusions. In: 18th AIAA/IEEE Digital Avionics Systems Conference, St. Louis (MO) (1999)
23. Roscoe, A.W.: Model-checking CSP. In: A Classical Mind, Eassys in Honour of C.A.R. Hoare. Prentice-Hall International (1997)
24. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall International, Upper Saddle River (1997)
25. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall International, Upper Saddle River (1998)
26. Roscoe, A.W.: Understanding Concurrent Systems, 1st edn. Springer, New York (2010)
27. Rushby, J.: Using model checking to help discover mode confusion and other automation surprises. In: Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD'99), Liege, Belgium (1999)

28. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. In: Reliability Engineering and System Safety (2002)
29. Sarter, N.B., Woods, D.D., Billings, C.E.: Automation surprises. In: Salvendy, G. (ed.) Handbook of Human Factors and Ergonomics. Wiley, New York (1997)
30. Schrönen, M.: Methodology for the Development of Microprocessor-Based Safety-Critical Systems. Monographs of the Bremen Institute of Safet Systems 8, Bremen University (1998). Shaker Verlag, Aachen
31. Schneider, S.: Concurrent and Real-Time Systems: The CSP Approach. Wiley, New York (1999)
32. Twele, L., Schlingloff, H., Szczerbicka, H.: Performability analysis of an avionics-interface. In: Proceedings of IEEE Conference on Systems, Man and Cybernetics, San Diego, N.J., pp. 499–504 (1998)
33. University of Oxford. FDR3 User Manual, FDR 3.4 edition. https://www.cs.ox.ac.uk/projects/fdr/manual/index.html

Part VIII
Web-Supported Communities in Science

Provably Correct Systems: Community, Connections, and Citations

Jonathan P. Bowen

Abstract The original European ESPRIT ProCoS I and II projects on *Provably Correct Systems* took place around a quarter of a century ago. Since then, the legacy of the initiative has spawned many researchers with careers in formal methods, forming a community of researchers with a common interest in this area. This chapter uses one of the leaders on the ProCoS initiative, Ernst-Rüdiger Olderog, as an example in demonstrating connections within and around the ProCoS research community. This is formalized using the Z notation to make the description more precise, especially with respect to collaborations undertaken through coauthorship of publications and subsequent citations to this research output. Matching visualizations of the relationships are included. The social science concept of a Community of Practice (CoP) is introduced in this context. Finally, consideration of citation metrics is also included.

1 Background

Historically, the creation of scientific knowledge has relied on collaborative efforts by successive generations through the centuries [39]. Scientific advances are gradually developed by a community of researchers over time (e.g., the abstract algebra of the French mathematician Évariste Galois (1811–1832) leading to Galois theory and group theory [11]). A scientific theory can be modelled as a mathematical graph of questions posed by scientists (represented by the vertices of the graph) and the corresponding answers (modelled by arcs connecting the vertices in the graph) [36]. The answers to questions lead to further questions and so the process continues, potentially ad infinitum. In general, mathematical logic underlies the valid reasoning that is required for worthwhile development of scientific theories and knowledge [20].

J.P. Bowen (✉)
School of Engineering, London South Bank University, Borough Road,
London SE1 0AA, UK
e-mail: jonathan.bowen@lsbu.ac.uk
URL: http://www.jpbowen.com

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_13

In recent years, the speed of transmission and the quantity of knowledge available has accelerated dramatically, especially with improvements in the Internet and specifically the increasing use of the World Wide Web [1]. Whereas previously academic papers were published on paper in journals, conference proceedings, technical reports, books, etc., now all these means of communication can and often are done largely electronically online. The plethora of information has also become indexed more and more effectively, especially with the advent of the PageRank algorithm as used by Google [30].

In this chapter, we use the European ProCoS ("Provably Correct Systems") initiative of the 1990s [2, 29] as an example of a foundational community of academic researchers working in various areas towards a common aim. We consider the related issue of the production of publications and their citations as an important aspect of scholarly activity. We model some aspects of this formally using the Z notation [5, 7, 37] to help in disambiguating some of the concepts that are often left somewhat nebulous in social science (e.g., with respect to a Community of Practice [40, 41]).

Section 2 introduces the European collaborative ProCoS projects and the subsequent Working Group of the 1990s. In Sect. 3, we present an example ProCoS researcher and their relationship with other researchers through coauthorship and citations, with visualizations of these relationships. The Section formalizes the relationship of researchers in an academic community such as that generated by ProCoS and Sect. 4 extends this to cover a formalized Community of Practice. Section 5 considers some of the citation metrics that are available for measuring a researcher's influence, including their shortcomings, using publication corpuses that are now available online. Finally Sect. 6 provides a conclusion and some possible future directions.

2 The ProCoS Community

In this section, we consider the development of the ProCoS initiative and the community that it has created. The seeds of the ProCoS projects on "Provably Correct Systems" took place in the 1980s [2, 29], coming out of the formal methods community [3, 12]. The CLInc Verified Stack initiative of Computational Logic Inc. in the USA [31, 42], using the Boyer-Moore Nqthm theorem proving to verify a linked set of hardware, kernel and software in a unified framework, was an inspiration for the initial ProCoS project. Whereas CLInc was a closely connected set of mechanically proved layers, ProCoS concentrated more on possible formal approaches to the issues of verifying a complete system at more levels from requirements, specification, design, and compilation, using a diverse set of partners around Europe with different backgrounds, expertise, and interests, but with a common overall goal. A ProCoS "tower" with appropriate formalisms and approaches was proposed to investigate proving a system correct in a linked way at the various levels of abstraction. The approach was based around the Occam parallel programming language and Transputer microprocessor architecture. A gas burner was used as a motivating example for much of the work.

The first ProCoS project was for $2\frac{1}{2}$ years (1989–1991) with seven academic partners [2]. The subsequent ProCoS II project (1992–1995) involved a more focused set of four academic partners [15]. Subsequently a ProCoS-WG Working Group of 25 partners (1994–1997) allowed a more diverse set of researchers to engage in the ProCoS approach, including industrial partners [16]. The entire ProCoS effort covered these and a number of other associated projects and initiatives [9].

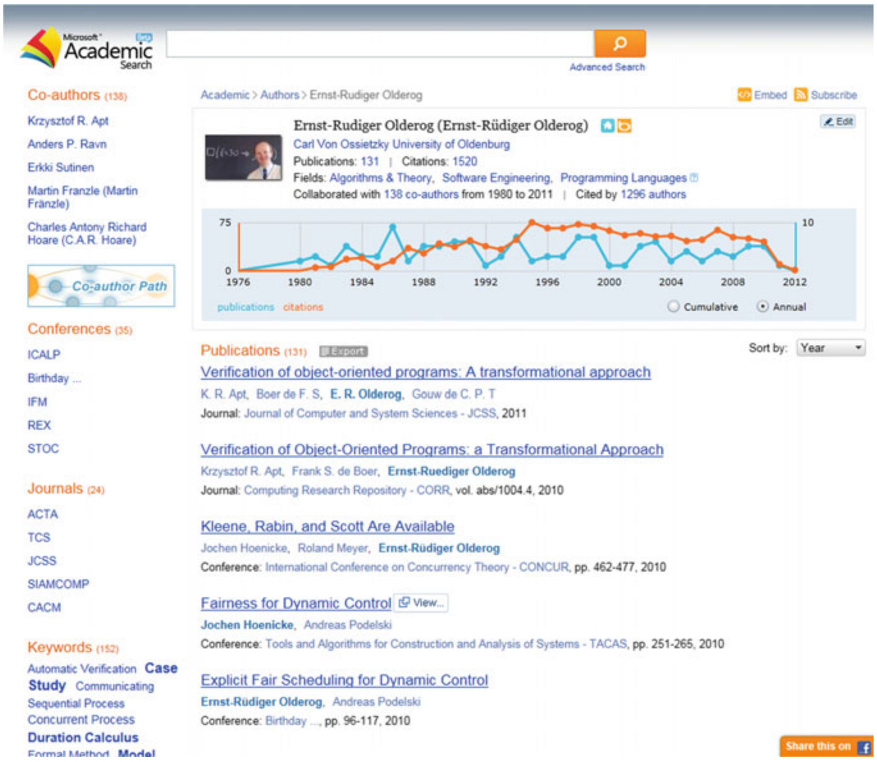
The ProCoS projects worked on various aspects of formal system development at different related levels of abstraction, including program compilation from an Occam-based programming language to a Transputer-based instruction set [10, 23, 29]. A gas burner was used extensively as a case study and this helped to inspire the development of Duration Calculus for succinctly formalized real-time requirements [43]. A novel provably correct compiling specification approach was also developed using a compiling relation for the various constructs in the language that could be proved using algebraic laws [27]. This was later extended to a larger language including recursion [21, 22]. The project used algebraic and operational semantics in its various approaches. The relationship between these and also denotational semantics was later demonstrated more universally in the *Unified Theories of Programming* (UTP) approach [26].

3 A Community Around a Researcher

Here we use the German computer scientist and one of the original leaders on the ProCoS project, Ernst-Rüdiger Olderog [32–34] of the University of Oldenburg, as an example of a leading member of a community of researchers, for illustrative purposes. Of course an endeavour like ProCoS has a number of leading researchers in practice, each with different influences, both within and outside the ProCoS community itself. All could be studied in a similar way, with differing characteristics in each case (e.g., see [6] for another example).

In the section, the visualization capabilities of the Microsoft *Academic Search* facility (available online under http://academic.research.microsoft.com) are used to illustrate a community around a particular researcher. This was initiated at the Microsoft Beijing research laboratory in China. As a starting point, see Fig. 1 for E.-R. Olderog's home page on the Academic Search website. The site's facilities include graphical presentation of direct relationships between collaborators as coauthors of publications, direct citations of other researchers to an individual's publications, and indirect connections between any two authors through intermediate coauthors in a transitive manner.

Academic Search also lists the coauthors, conferences and journals for each author, in reverse order of publication count, and the main keywords associated with the publications of an author (see Fig. 1). For example, three out of the top five coauthors of E.-R. Olderog were associated with the ProCoS project. In addition, he is particularly active in the *International Colloquium on Automata, Languages, and Programming* (ICALP), the *Integrated Formal Methods* (IFM) conferences, as



Co-authors (138)
Krzysztof R. Apt
Anders P. Ravn
Erkki Sutinen
Martin Franzle (Martin Fränzle)
Charles Antony Richard Hoare (C.A.R. Hoare)

Conferences (35)
ICALP
Birthday ...
IFM
REX
STOC

Journals (24)
ACTA
TCS
JCSS
SIAMCOMP
CACM

Keywords (152)
Automatic Verification Case Study Communicating Sequential Process Concurrent Process Duration Calculus Formal Method Model

Academic > Authors > Ernst-Rudiger Olderog

Ernst-Rudiger Olderog (Ernst-Rüdiger Olderog)
Carl Von Ossietzky University of Oldenburg
Publications: 131 | Citations: 1520
Fields: Algorithms & Theory, Software Engineering, Programming Languages
Collaborated with 138 co-authors from 1980 to 2011 | Cited by 1296 authors

Publications (131)

Verification of object-oriented programs: A transformational approach
K. R. Apt, Boer de F. S, E. R. Olderog, Gouw de C. P. T
Journal: Journal of Computer and System Sciences - JCSS, 2011

Verification of Object-Oriented Programs: a Transformational Approach
Krzysztof R. Apt, Frank S. de Boer, Ernst-Ruediger Olderog
Journal: Computing Research Repository - CORR, vol. abs/1004.4, 2010

Kleene, Rabin, and Scott Are Available
Jochen Hoenicke, Roland Meyer, Ernst-Rüdiger Olderog
Conference: International Conference on Concurrency Theory - CONCUR, pp. 462-477, 2010

Fairness for Dynamic Control
Jochen Hoenicke, Andreas Podelski
Conference: Tools and Algorithms for Construction and Analysis of Systems - TACAS, pp. 251-265, 2010

Explicit Fair Scheduling for Dynamic Control
Ernst-Rüdiger Olderog, Andreas Podelski
Conference: Birthday ..., pp. 96-117, 2010

Fig. 1 Publication and citation statistics for Ernst-Rüdiger Olderog on Academic Search

well as the *Acta Informatica* and *Theoretical Computer Science* journals (again, see Fig. 1), Important keywords include "Duration Calculus", a direct (and unpredicted) result of the ProCoS project.

The links between coauthors and citing authors form mathematical graphs [14]. These can be modelled using relations. The Z notation [24, 37] is a convenient notation to present these formally, as previously demonstrated in [6], since relations are an important aspect of the language and are easily represented. Here we concentrate on authors, rather than individual publications, and the paths of coauthors that connect researchers. In particular, we augment this model to consider the "collaborative distance" (the length of the shortest path) between an arbitrary pair of authors in terms of transitive coauthorship. We model all the possible paths between such authors as a set of sequences of authors where the two authors under consideration are the first and last author in each of the sequences. The two authors also do not occur within these sequences and authors are not repeated in the sequences either.

We use the concept of graphs in our mathematical modelling. A general graph can be modelled as a relation in Z, using a generic constant on any set X:

$$[X]$$
$$graph : X \leftrightarrow X$$

We can refine a general graph and consider a model for an undirected graph in Z:

$$[X]$$
$$ugraph : \mathbb{P}\, graph$$
$$\begin{array}{l} ugraph = ugraph^{\sim} \\ ugraph \cap \mathrm{id}\, X = \varnothing \end{array}$$

Here all nodes (authors) are connected in both directions (as coauthors) and also a node cannot be connected to itself (i.e., an author cannot be a coauthor with themselves). In the above definition, "$^{\sim}$" indicates the inverse of a relation and "id" produces the identity relation from a set.

Academic communities consist of people that have authored publications. In Z, this can be modelled as a given set:

$$[PEOPLE]$$

In an academic community of researchers for a particular area, there is often a main key researcher leading the field's publications. Then there is a wider number of researchers that have published papers in the field. Typically published works have a number of coauthors. Published authors may be related to other authors transitively through coauthorship. Authors may also be cited by other published authors, even if not related through coauthorship. These relationships can be modelled formally using graphs:

$$Researchers$$
$$\begin{array}{l} main : PEOPLE \\ published : \mathbb{F}_1\, PEOPLE \\ coauthors, related, citing\_authors : PEOPLE \leftrightarrow PEOPLE \end{array}$$
$$\begin{array}{l} main \in published \\ coauthors \subseteq ugraph[published] \\ related \subseteq ugraph[published] \\ related = coauthors^{+} \\ citing\_authors \subseteq graph[published] \end{array}$$

Note that "$\mathbb{F}_1$" indicates a finite non-empty set and "$^{+}$" indicates irreflexive transitive closure above.

The Academic Search facility enables graphical visualization of the coauthors (e.g., see Fig. 2) and citing authors (e.g., see Fig. 3) for any particular *author* in its database. Figure 2 provides a pictorial view of a subset of the relation $\{author\} \lhd related \rhd coauthors(\!|\ \{author\}\ |\!)$ (where "$\lhd$" indicates domain restriction of a

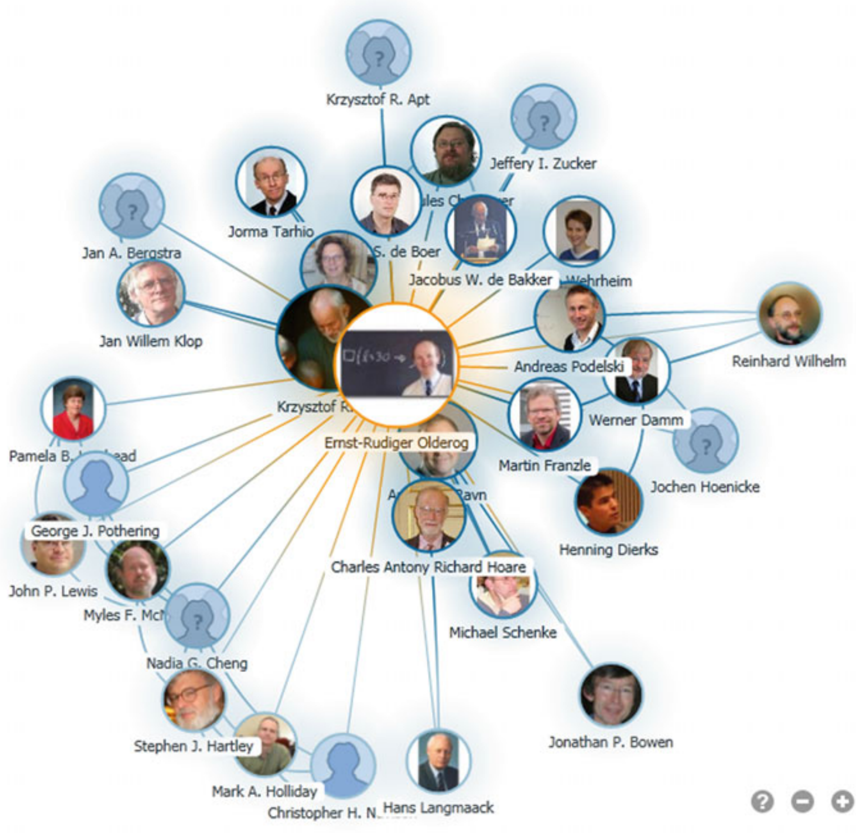


Fig. 2 Primary coauthors of Ernst-Rüdiger Olderog on Academic Search

relation, "$\rhd$" indicates range restriction of a relation, and "$(\!| \ldots |\!)$" indicates a relational image of a subset of the domain) for a specific *author* (in this case E.-R. Olderog) at the centre. Connections between coauthors who have themselves written publications together can be shown as well, in addition to coauthorship with the main author under consideration. This results in groupings of coauthors that are interconnected in a way than can be seen visually very quickly. For example, in this case all the coauthors associated with the ProCoS project are in the lower right-hand quadrant, including the author of this chapter.

Figure 3 gives a partial pictorial view of the relation $\{author\} \lhd citing\_authors$, again for a specific *author* located at the top left position in the diagram. Citations from authors involved with the ProCoS project are largely grouped on the left-hand side of the diagram, during Olderog's early career. Later citations are to the right.

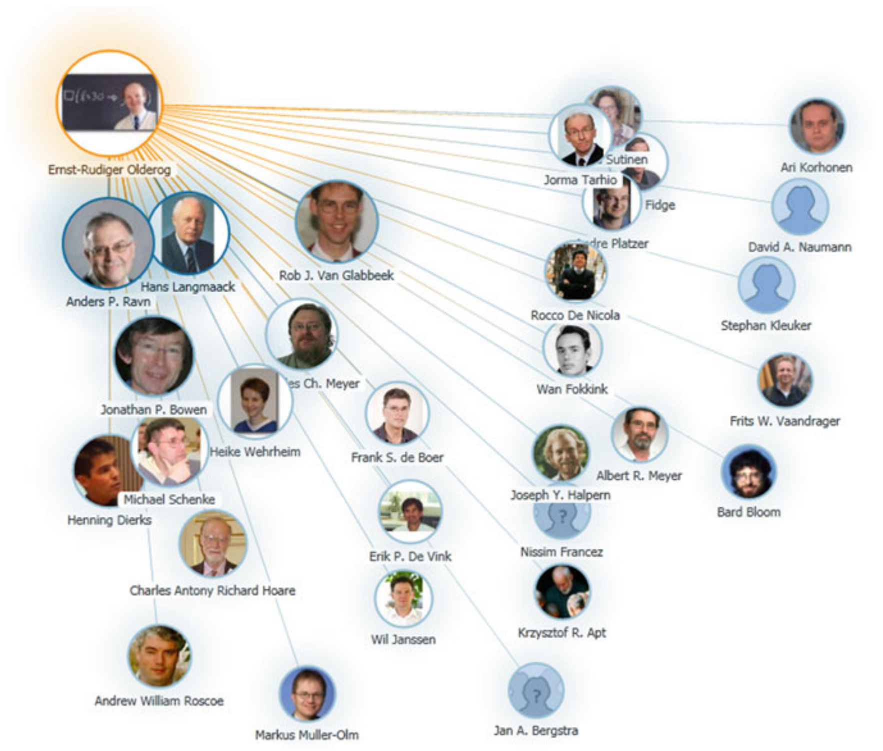


Fig. 3 Primary citing authors for Ernst-Rüdiger Olderog on Academic Search

Next we consider paths between pairs of nodes (authors):

$$
\begin{array}{|l}
[X] \\
\hline
path : (X \times X) \leftrightarrow \mathrm{iseq}\, X \\
\hline
\forall x_1, x_2 : X;\ s : \mathrm{iseq}\, X \mid \# s > 1 \bullet \\
\quad (x_1, x_2) \mapsto s \in path \Leftrightarrow \\
\qquad head\ s = x_1 \wedge \\
\qquad last\ s = x_2 \wedge \\
\qquad (\forall n : \mathbb{N}_1 \mid n < \# s \bullet (s\ n, s(succ\ n)) \in graph) \\
\hline
\end{array}
$$

The paths are modelled as injective sequences ("iseq") of length more than one, where the first and last entries in the sequences are the two nodes under consideration and all adjacent pairs in the sequence are directly connected in the graph. Because the sequences are injective, no nodes are repeated in these sequences. This means that the pair of nodes under consideration are always two different nodes.

The collaborative distance of two authors can be of particular interest. Two authors may be connected in many different ways by sequences of coauthors or even in no way whatsoever (effectively an infinite collaborative distance). The shortest (minimum) connection between two different authors is of special interest.

$$
\begin{array}{|l}
[X] \\
\hline
dist : X \times X \mapsto \mathbb{N}_1 \\
\hline
\forall x_1, x_2 : X \mid (x_1, x_2) \in \mathrm{dom}\, path \bullet \\
\qquad dist(x_1, x_2) = min(\#\llparenthesis\, path \llparenthesis \{ (x_1, x_2) \} \rrparenthesis \rrparenthesis) \\
\hline
\end{array}
$$

In recent years, the "Erdős number" (i.e., the collaborative distance from Erdős) has become a metric for involvement in mathematical and even computer science research [14]. Paul Erdős, a very collaborative 20th century mathematician, is considered to have an Erdős number of 0. His direct coauthors (511 of them) have an Erdős number of 1. Other authors can be assigned a number that is the minimum length of the coauthorship path that links them with Erdős, assuming there is such a path. More generally, considering a main author, the collaborative distance of other authors from the main author can be considered, or indeed between any arbitrary pair of published authors. Authors who have written publications with coauthors of Erdős (the main author) but not with Erdős himself have an Erdős number of 2. This process can be continued in an iterative manner, using a path of minimum length to determine the Erdős number when there is more than one path, as is typically the case for active researchers in the field.

Academic Search can provide a graphical view of a number of the shortest paths between any two coauthors, with the Hungarian mathematician and prolific paper coauthor Paul Erdős (1913–1996) provided as the standard second author unless a different author is explicitly selected. Figure 4 shows an example for E.-R. Olderog. Here, five paths with a collaborative distance of four are shown. The five researchers on the right directly connected to Erdős have an Erdős number of 1. Of the five researchers directly connected to Olderog on the left, one (C.A.R. Hoare) was also on the ProCoS project. Of course the database of authors and publications may not be complete or accurate (e.g., especially for authors with common names) and there could be shorter paths between two authors in practice.

4 Community of Practice

A Community of Practice (CoP) [40, 41] is a widely accepted social science approach used as a framework in the study of the community-based process of producing a particular Body of Knowledge (BoK) [13]. An example of a CoP is that generated by the ProCoS initiative in the area of provably correct systems [10, 23]. The important elements of a CoP include a domain of common interest (e.g., provably correct systems), a community willing to engage with each other (e.g., members of the

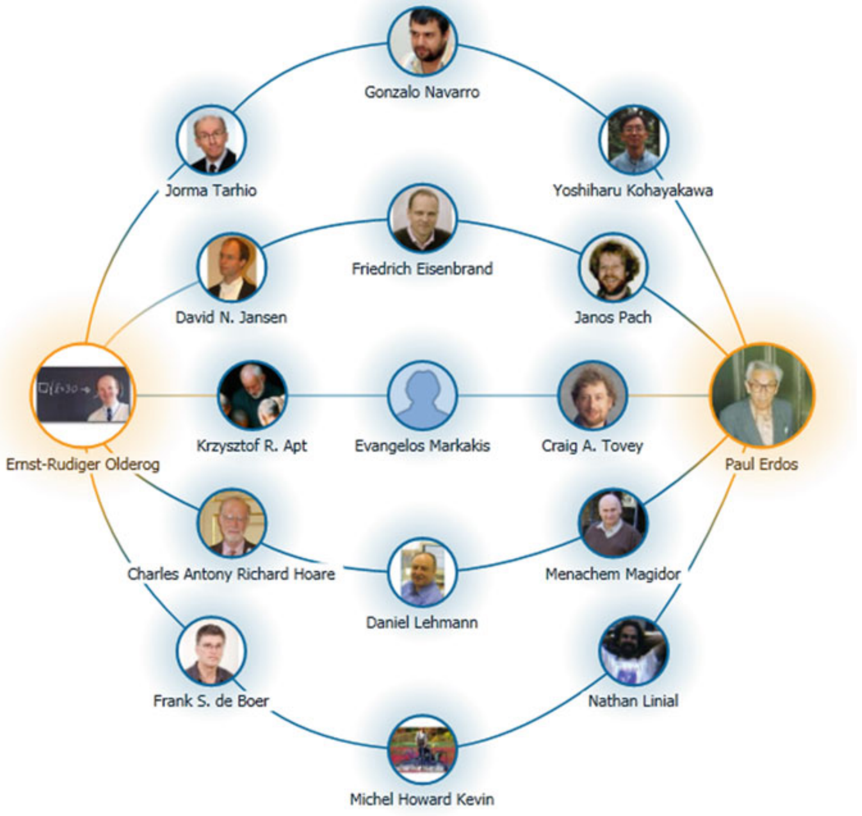


Fig. 4 A selection of connections with Paul Erdős for Ernst-Rüdiger Olderog on Academic Search

ProCoS projects and Working Group), and exploration of new knowledge to improve practice (e.g., Duration Calculus [43] and later UTP [26]).

Communities of Practice may be overlapping or subsets of other CoPs. The *main* author, as introduced earlier, could be considered as a coordinator of a Community of Practice. Direct coauthors with the main coordinator typical take on a major organizational and editorial role in the CoP. Those that are related to the main author by transitive coauthorship are active members. These people form the core of the CoP membership. Those that cite any of the above are peripheral members of the CoP. Finally, other unrelated published authors are considered to be outsiders to the CoP, but are potential members.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\; CoP \;}} \\
Researchers \\
editorial, active, core, peripheral, cop, outsiders : \mathbb{F}\, PEOPLE \\
\hline
editorial = coauthors \,(\!|\, \{main\} \,|\!)\, \\
active = related \,(\!|\, \{main\} \,|\!)\, \setminus editorial \\
core = \{main\} \cup editorial \cup active \\
peripheral = citing\_authors \,(\!|\, core \,|\!)\, \setminus core \\
cop = core \cup peripheral \\
outsiders = published \setminus cop \\
\hline
\end{array}
$$

In the context of the ProCoS example based on E.-R. Olderog as the main author nd leader at one of the collaborating sites, those related by transitive authorship could be considered core members. The collaborative distance could be limited to some set maximum if desired. Authors that have cited core ProCoS researchers are peripheral members of the ProCoS community. All other published researchers are considered outsiders to the community. Of course this formalization could be varied if desired. For example, the maximum collaborative distance from the "main" author for *core* members could be set. However, whatever formalization is chosen, this gives a precise definition for an informal social science concept of a CoP, potentially allowing a more rigorous discussion about the nature of a CoP.

5 Citation Metrics

In the previous two section we considered published authors and their communities of researchers. Here we consider individual authors and their publications. Nowadays there are various web-based databases that index academic publications online, including facilities that allow citation data to be calculated automatically. For example, Google has a specific search facility for indexing scholarly publications through *Google Scholar* (http://scholar.google.com). Books are also available online through *Google Books* (http://books.google.com), although this does not record citation information. Google Scholar has very complete and up-to-date information compared to other sources [18], even if this can mean it is less reliable and authoritative due to the lack of human checking. However, Google Scholar provides a facility for individuals to generate a personalized and publicly available web page presenting their own publications with citation information that can be hand-corrected by the author involved as needed at any time.

The automated search through crawling of websites including publications with references that is undertaken by Google Scholar is fairly reliable for publications with a reasonable number of citations. The various citations allows automated improvement of the information. Typically for a given author on their personalized page, the publications list includes a "long tail" of uncited or lesser cited publications, some of which can be spurious and with poor default information. These can be edited or deleted as required. In addition to valid publications, Google also trawls online programme committee data for conferences, In these cases all the committee members are normally considered to be authors by Google Scholar.

There are various possible ways to measure the influence of a researcher through their publications. One of the simplest is the number of citations. This can vary widely between disciplines, and of course depends on the length of the career so far for a researcher, as well as patterns of collaboration with other researchers. Joint publications mean that a researcher can appear much more productive than if only single-author publications are produced. Thus the sciences where multi-authored papers are the norm fair better for citation counts than the humanities where single-author books on research are more normal. However within a given discipline (e.g., computer science), comparison using citation metrics has some validity.

The total number of citations can be deceptive for reasons dependent on the field. For researchers with a reasonable number of publications, there is a standard pattern to the distribution of citations for individual publications [17]. Normally a researcher has a small number of publications with significant numbers of citations (and thus influence). Conversely there is typically a much larger number of publications with only a few citations (and hence much less influence). In practice, the small number of highly cited publications are much more important in terms of influence than the larger number of lesser-cited publications. Yet the total number of citations for the latter may be significant in size compared with the former.

To overcome these issues, further citations metrics than just citation counts have been developed. One of the most popular is the *h-index* [25]. This measures the number h of publications by an individual author that have h or more citations. This provides a reasonably simple measure of the influence of an author through their most highly cited publications. All other lesser-cited publications have no influence on this metric. Google Scholar includes this metric on personal pages generated by individual researchers automatically,

The h-index can be formalized using the Z notation [5, 37], for example. This was done in a functional style in an earlier paper [6]. Here we present a more relational and arguably more abstract definition. As in the previous paper, we use a Z "bag" (sometimes also called a multiset) to model the citation count for each individual publication. We use a generic definition for flexibility.

$$[X]$$

$$\text{h-index} : \text{bag}\, X \rightarrow \mathbb{N}$$

$$\forall b : \text{bag}\, X;\ h : \mathbb{N} \bullet \text{h-index}\, b = h \Leftrightarrow h = \#\{x : X \mid b(x) \geq h\}$$

Note that Z bags are defined as $\text{bag}\, X == X \twoheadrightarrow\!\!\!\!\!\!\!+\ \mathbb{N}_1$, a partial function from any generic set X to non-zero natural numbers. X can be used to represent cited publications, for example, mapped to the number of citations associated with each of these publications. A publication with no citations will not be covered in this mapping, The h-index metric should be treated with some caution since comparison across different academic disciplines and historical periods may well not be valid due to differences in patterns of publication. Some researchers produce a very small number of highly influential papers. Alan Turing (1912–1954) is an example of such a researcher, with three extremely important papers, each founding a field (theoretical computer science, Artificial Intelligence, and mathematical biology) and new associated communities of researchers [8]. He was also a lone researcher will mostly single-author papers and including few references. In addition to such issues, language is an important fact and non-English publications tend to fare less well in the automated generation of such data, which are typically undertaken by English-speaking project teams.

In humanities, single-author publications are the norm, as previously mentioned. In contemporary computer science, a small number of coauthors is typical (e.g., two to three on average), with acknowledgements to others that have helped with the research in some smaller way. A supervisor may be named as second author to publication by a doctoral student, whereas in humanities the supervisor may well not be named. In chemistry, a larger number of coauthors is typical, with a team of people (e.g., ten or more) working on a problem, providing different expertise. Indeed, coauthors may not have been involved in writing the paper at all, but may have given help with an experiment, for example. In physics, very large numbers of coauthors are possible for sizable and expensive initiatives (perhaps even hundreds, e.g., experiments at CERN).

Many papers on the ProCoS projects were collaborative, including multi-site and multi-country collaboration. Indeed, this was an important aspect of the initiative to encourage such collaboration across Europe. Nowadays a record of such collaboration is readily available online through comprehensive facilities such as Google Scholar. Individual researchers can add links to coauthors that also have personal Google Scholar pages and these are suggested by the system if a coauthor creates a new personal Google Scholar page. E.-R. Olderog has 23 such coauthors (https://scholar.google.com/citations?user=G57CATkAAAAJ).

Figure 5 shows a graph of the citations E.-R. Olderog's publications by year on Google Scholar, from 1982 to the present. The ProCoS I/II projects and the ProCoS Working Group took place from 1989 to 1997 and this was a period of increasing citations for Olderog. Soon afterwards, citations dropped off quite rapidly from 1998 and have only recovered to previous levels very recently to exceed these in 2015. This may indicate that the period of the main activity of ProCoS was a highly productive one with respect citations and thus research influence for Olderog.

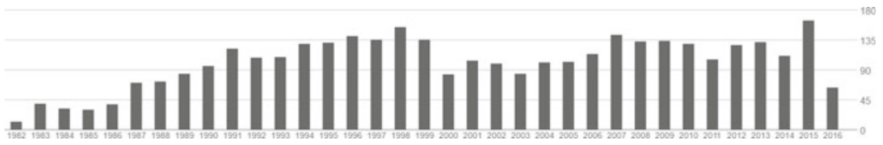


Fig. 5 Citations of Ernst-Rüdiger Olderog by year on Google Scholar (1982–2016)

On an individual author's personalized Google Scholar page, as set up and editable by the author, the number of citations for each publication and the total sum of citations together with the author's h-index and also *i10-index* (the number of publications with ten or more citations [6]), are displayed, for the last six years and for all time. A particular aspect that is lacking in Google Scholar is any significant visualization facility. The only visual output provided is in the form of bar charts of the number of citations each year for authors and also for individual papers. This is useful but not very impressive.

As an alternative to Google Scholar, Microsoft Research's *Academic Search* (see http://academic.research.microsoft.com) provides another online database of academic publications. Unfortunately the resource is by no means as complete or up to date as the information provided by Google Scholar, although historical coverage of journals in the sciences is good. It appears that regular updates ceased in 2012. On the positive side, Academic Search does provide much better visualization facilities compared to Google Scholar, as illustrated in Sect. 3. It has also been possible for any individual to submit corrections regarding any publication entry within the database. These have been checked by a human before being accepted (after some variable delay). Note that Microsoft is replacing Academic Search with a more mainstream facility, *Microsoft Academic* (https://academic.microsoft.com).

In addition to the h-index, Academic Search also provides the "*g-index*" [19] for each author. This is a refinement of the h-index and arguably provides a somewhat improved indication of an author's academic influence. The g-index measure gives very highly cited publications (e.g., a significant book or foundational paper) more weight than with the h-index, where additional citations over and above the h-index itself for individual publications have no effect on its value. In the case of g-index, the most cited g papers must have at least g^2 citations in combination. Thus very highly cited publications do contribute additional weight to the g-index. Indeed, the value of the g-index is always at least as great as the h-index for a given author and is greater if there are some very highly cited publications.

In [6], the g-index was formally defined in Z using a functional style, close to how its calculation could be implemented. Here we use a more relational style of specification, arguably more abstract and certainly less easily directly implemented in an imperative programming language:

$$\begin{array}{l} [X] \\ \hline \text{g-index} : \text{bag}\,X \rightarrow \mathbb{N} \\ \hline \forall b : \text{bag}\,X;\ g : \mathbb{N} \bullet \\ \quad g * g \leq max\{a : \text{bag}\,X \mid a \subseteq b \land \#a = g \bullet \Sigma a\} < (g+1) * (g+1) \\ \hline \end{array}$$

Note that the Σ function calculates the sum of all items in a bag and was defined formally in [6].

Other citation indices include the i10-index as used on Google Scholar, indicating the number of publications with ten or more citations [6] and the lesser used "*f-index*" [28], designed to be fairer in determining researchers with influence across more communities. With a plethora of citation indices, caution should be taken as to their reliability in practice. Encouraging the production of more papers with incremental results can be detrimental to the advancement of scientific knowledge [35].

6 Conclusion

This chapter has presented the collaborative European ESPRIT ProCoS projects and Working Group on Provably Correct Systems of the 1990s and the community that this formed. It considers the framework of a Community of Practice (CoP) in the context of collaboration and influence within such a community through coauthorship. We have also considered citations to individual publications for a particular author. The development of knowledge depends on such communities of researchers, which are created and then transmogrify as needed, depending on the interests of individual researchers interacting in the larger community.

A case study of an individual involved with the ProCoS project has been included with visualization of connections between researchers. Key concepts have been formalized using the Z notation. Further formalizations and considerations of sociological issues within the CoP framework could be considered in more detail in the future.

As well as communities of researchers, this chapter has discussed citation metrics for individual researchers, which have become increasingly widespread. It should be noted that the relevance of these, like most metrics, is a matter of debate and any such measurements should always be treated with caution and interpreted in an appropriate manner. In particular, the citations at any particular point in time are a snapshot with no precise indication of future citations. In addition, general concepts are often not cited as all. Many disciplines have a practice of including "passive" authors that have not directly undertaken the research, perhaps acting as a supervisor or funder instead. These and other issues mean that all citation statistics should be used with caution.

Possible future directions include considering the graphs of relationships between authors and publications more holistically to model movements and influences, but this is beyond the scope of this current chapter.

Acknowledgements Jonathan Bowen is grateful for financial support from Museophile Limited. Thank you to Microsoft for the Academic Search facility, which provided the screenshots for the figures in this chapter. Many thanks to collaborators on the ProCoS projects and Working Group during the 1990s and subsequently. A special thank you to Prof. Dr Ernst-Rüdiger Olderog of Oldenburg University for the individual case study. The Z notation in this chapter has been formatted using the `oz.sty` LATEX style and type-checked using the *f*UZZ type-checker [38]. Finally, the reviewers provided helpful comments that improved the presentation and content of the chapter.

References

1. Berners-Lee, T.: Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web. HarperCollins, New York (2000)
2. Bjørner, D., Hoare, C.A.R., Bowen, J.P., He, J., Langmaack, H., Olderog, E.R., Martin, U.H., Stavridou, V., Nielson, F., Nielson, H.R., Barringer, H., Edwards, D., Løvengreen, H.H., Ravn, A.P., Rischel, H.S.: A ProCoS project description. Bull. Eur. Assoc. Theor. Comput. Sci. (EATCS) **39**, 60–73 (1989). Oct
3. Boca, P.P., Bowen, J.P., Siddiqi, J. (eds.): Formal Methods: State of the Art and New Directions. Springer, Heidelberg (2010)
4. Bowen, J.P. (ed.): Towards Verified Systems, Real-Time Safety Critical Systems, vol. 2. Elsevier, Amsterdam (1994)
5. Bowen, J.P.: Z: A formal specification notation. In: Frappier, M., Habrias, H. (eds.) Software Specification Methods: An Overview Using a Case Study, chap. 1, pp. 3–19. FACIT series, Springer, Heidelberg (2001)
6. Bowen, J.P.: A relational approach to an algebraic community: From Paul Erdős to He Jifeng. In: Liu, Z., Woodcock, J.C.P., Zhu, H. (eds.) Theories of Programming and Formal Methods. Lecture Notes in Computer Science, vol. 8051, pp. 54–66. Springer, Heidelberg (2013)
7. Bowen, J.P.: The Z notation: whence the cause and whither the course? In Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems: First International School, SETSS 2014, Chongqing, China, September 8–13, 2014. Lecture Notes in Computer Science, vol. 9506, pp. 104–151. Springer, Heidelberg (2016)
8. Bowen, J.P.: Alan Turing: virtuosity and visualisation. In Bowen, J.P., Diprose, G., Lambert, N. (eds.) EVA London 2016 Conference Proceedings. Electronic Workshops in Computing (eWiC), pp. 197–204. BCS (2016)
9. Bowen, J.P.: ProCoS – Provably Correct Systems. Formal Methods Wiki, Wikia. http://formalmethods.wikia.com/wiki/ProCoS. (Accessed June 2016)
10. Bowen, J.P., Fränzle, M., Olderog, E.R., Ravn, A.P.: Developing correct systems. In: Proceedings of the 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland. pp. 176–189. IEEE Computer Society Press, Washington (1993)
11. Bowen, J.P., Giannini, T.: Galois connections: mathematics, art, and archives. In: Ng, K., Bowen, J.P., Lambert, N. (eds.) EVA London 2015 Conference Proceedings, pp. 176–183. Electronic Workshops in Computing (eWiC), BCS (2015)
12. Bowen, J.P., Hinchey, M.G.: Formal methods. In: Gonzalez, T.F., Diaz-Herrera, J., Tucker, A.B. (eds.) Computing Handbook, vol. 1, 3rd edn., Chap. 71, pp. 1–25. CRC Press, Florida (2014)
13. Bowen, J.P., Reeves, S.: From a community of practice to a body of knowledge: a case study of the formal methods community. In: Butler, M., Schulte, W. (eds.) FM 2011: 17th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 6664, pp. 308–322. Springer, Heidelberg (2011)
14. Bowen, J.P., Wilson, R.J.: Visualising virtual communities: From Erdős to the arts. In: Dunn, S., Bowen, J.P., Ng, K. (eds.) EVA London 2012 Conference Proceedings. pp. 238–244. Electronic Workshops in Computing (eWiC), BCS (2012). arXiv:1207.3420v1
15. Bowen, J.P., et al.: A ProCoS II project description: ESPRIT Basic Research project 7071. Bull. Eur. Assoc. Theor. Comput. Sci. (EATCS) **50**, 128–137 (1993)
16. Bowen, J.P., et al.: A ProCoS-WG Working Group description: ESPRIT Basic Research 8694. Bull. Eur. Assoc. Theor. Comput. Sci. (EATCS) **53**, 136–145 (1994). Jun
17. Breuer, P.T., Bowen, J.P.: Empirical patterns in Google Scholar citation counts. In: Proceedings of the IEEE 8th International Symposium on Service Oriented System Engineering (SOSE), Cyberpatterns 2014: Third International Workshop on Cyberpatterns. pp. 398–403. IEEE Computer Society Press, Washington (2014). arXiv:1401.1861 [cs.DL]
18. Brezina, V.: Use of Google Scholar in corpus-driven EAP research. J. Engl. Acad. Purp. **11**(4), 319–331 (2012). Dec

19. Egghe, L.: Theory and practise of the g-index. Scientometrics **69**(1), 131–152 (2006)
20. Harré, R.: The Philosophies of Science: An Introductory Survey. Oxford University Press, Oxford (1972)
21. He, J.: Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers. International Series in Software Engineering. McGraw-Hill, New York (1995)
22. He, J., Bowen, J.P.: Specification, verification and prototyping of an optimized compiler. Form. Asp. Comput. **6**(6), 643–658 (1994)
23. He, J., Hoare, C.A.R., Fränzle, M., Müller-Olm, M., Olderog, E.R., Schenke, M., Hansen, M.R., Ravn, A.P., Rischel, H.: Provably correct systems. In: Langmaack, H., de Roever, W.P., Vytopil, J. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 863, pp. 288–335. Springer, Heidelberg (1994)
24. Henson, M.C., Reeves, S., Bowen, J.P.: Z logic and its consequences. CAI Comput. Inf. **22**(4), 381–415 (2003)
25. Hirsch, J.E.: An index to quantify an individual's scientific research output. In: Proceedings of the National Academy of Sciences **102**(46), 16569–16572 (2005). arXiv:physics/0508025
26. Hoare, C.A.R., He, J.: Unifying Theories of Programming. International Series in Computer Science. Prentice Hall, New Jersey (1998)
27. Hoare, C.A.R., He, J., Bowen, J.P., Pandya, P.K.: An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language. In: ESPRIT'90 Conference Proceedings, Brussels. pp. 804–818. CEC DG XIII (1990)
28. Katsaros, D., Akritidis, L., Bozanis, P.: The f index: quantifying the impact of coterminal citations on scientists' ranking. J. Assoc. Inf. Sci. Technol. **60**(5), 1051–1056 (2009). May
29. Langmaack, H., Ravn, A.P.: The ProCoS project: provably correct systems. In: Bowen [4], pp. 249–265, appendix B
30. Lanville, A.N., Meyer, C.D.: Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press, Princeton
31. Moore, J.S., et al.: Special issue on system verification. J. Autom. Reas. **5**(4), 409–530 (1989). Dec
32. Olderog, E.R.: Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship. Cambridge University Press, Cambridge (1991)
33. Olderog, E.R.: Interfaces between languages for communicating systems. In: Kuich, W. (ed.) Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 623, pp. 641–655. Springer, Heidelberg (1992). (invited paper)
34. Olderog, E.R. (ed.): Programming Concepts, Methods and Calculi, IFIP Transactions, vol. A-56. North-Holland (1994)
35. Parnas, D.L.: Stop the numbers game. Commun. ACM **50**(11), 19–21 (2007). Nov
36. Sanitt, N.: Graph theory. In: Science as a Questioning Process, Chap. 3, pp. 31–49. Institute of Physics Publishing (1996)
37. Spivey, J.M.: The Z Notation: A reference manual. Prentice Hall (1989/1992/2001). http://spivey.oriel.ox.ac.uk/mike/zrm/
38. Spivey, J.M.: The *f*UZZ type-checker for Z. Technical report, University of Oxford, UK (2008). http://spivey.oriel.ox.ac.uk/mike/fuzz/
39. Van Doren, C.: A History of Knowledge: Past, Present, and Future. Ballantine Books (1991)
40. Wenger, E.: Communities of Practice: Learning, Meaning, and Identity. Cambridge University Press, Cambridge (1998)
41. Wenger, E., McDermott, R.A., Snyder, W.: Cultivating Communities of Practice: A guide to managing knowledge. Harvard Business School Press, Massachusetts (2002)
42. Young, W.D.: System verification and the CLI stack. In: Bowen [4], pp. 225–248, appendix A
43. Zhou, C., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991). Dec

Erratum to: ProCoS: How It All Began – as Seen from Denmark

Dines Bjørner

Erratum to: ProCoS: How It All Began – as Seen from Denmark in: M. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_1

The original version of the book was inadvertently published with old manuscript instead of revised manuscript for Chapter 1. The erratum chapter and the book have been updated with the change.

The updated online version for this chapter can be found at http://dx.doi.org/10.1007/978-3-319-48628-4_1

M.G. Hinchey et al. (eds.), *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, DOI 10.1007/978-3-319-48628-4_14