Natarajan Shankar
Jim Woodcock (Eds.)

Verified Software: Theories, Tools, Experiments

LNCS 5295

Second International Conference, VSTTE 2008
Toronto, Canada, October 2008
Proceedings

Lecture Notes in Computer Science 5295

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

Natarajan Shankar Jim Woodcock (Eds.)

Verified Software: Theories, Tools, Experiments

Second International Conference, VSTTE 2008
Toronto, Canada, October 6-9, 2008
Proceedings

Springer

Volume Editors

Natarajan Shankar
SRI International
Computer Science Laboratory
MS EL256, 333 Ravenswood Avenue
Menlo Park, CA 94025-3493, USA
E-mail: shankar@csl.sri.com

Jim Woodcock
University of York
Department of Computer Science
Heslington, York YO10 5DD, UK
E-mail: jim@cs.york.ac.uk

Library of Congress Control Number: 2008935491

CR Subject Classification (1998): B.6.3, B.2.2, D.2.4, D.4, F.3, G.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-87872-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-87872-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12532179 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at the Second Working Conference on Verified Software: Theories, Tools, and Experiments held in Toronto during October 6–9, 2008. This followed a successful working conference held in Zurich in 2005, also published in *Lecture Notes in Computer Science* as volume 4171 (DOI 10.1007/978-3-540-69149-5). The second conference formally inaugurated the Verified Software Initiative (VSI), a 15-year, co-operative, international project directed at the scientific challenges of large-scale software verification. The scope of the cooperative effort includes the sharing and interoperability of tools, the alignment of theory and practice, the identification of challenge problems, the construction of benchmark suites, and the execution of large-scale experiments. The conference was open to everyone interested in participating actively in the VSI effort.

The scope of the VSTTE conferences includes all aspects of verified software, covering theoretical as well as experimental work:

- requirements modelling
- specification languages
- specification case studies
- formal calculi
- programming languages
- language semantics
- software design methods
- software testing
- automatic code generation
- refinement methodologies
- type systems
- computer security
- static analyzers
- dynamic analyzers
- model checkers
- theorem provers
- satisfiability checkers
- benchmarks
- challenge problems
- integrated verification environments

The conference was addressed by four keynote speakers:

- John Reynolds (Carnegie Mellon University)
- Moshe Vardi (Rice University)
- Andreas Podelski (University of Freiburg)
- Sriram Rajamani (Microsoft Research)

Two invited tutorials were given by:

- Eric Hehner (University of Toronto) *Practical Predicative Programming Primer*
- Ernie Cohen (Microsoft Research) *The Hyper-V Project*
- Leonardo de Moura (Microsoft Research) *SMT@Microsoft*

The volume contains 16 rigorously refereed papers on different topics covering the spectrum from theoretical results to verification experience reports. The conference also included a session of short presentations of ongoing work.

The main VSTTE 2008 conference hosted three specialized workshops on Theories, Tools, and Experiments for Verified Software.

VS-THEORY: Workshop on Theory for Verified Software

Dave Naumann (Stevens Institute)
Peter O'Hearn (Queen Mary, University of London)

Summary: Program verification has seen a worldwide renaissance, with many ongoing practical tool projects and experimental verification efforts. The current state of the field builds on fundamental theoretical advances of the past. Similarly, future advances on software verification will depend on developments in theory. This can range from the difficult and essential study of soundness of delicate proof methods, to the discovery of new specification techniques and proof methods, to dramatic simplification or unification of existing methods, to as yet unknown breakthroughs. The Verified Software Initiative (VSI) is envisaged as a 15-year Grand Challenge project to advance the state of software verification. Specific milestones and challenges of the VSI should often be concrete in nature, but advances beyond immediate progress will again depend on theoretical insights. The purpose of this workshop was to bring together theory and programming language researchers to discuss scientific challenges posed by software verification.

VS-TOOLS: Workshop on Tools in Verified Software

Daniel Kroening (University of Oxford)
Tiziana Margaria (University of Potsdam)

Summary: The scope of the workshop included submissions of technical and position papers on all aspects of tools conducted relating to verified software. Paper-and-pencil proofs are error-prone and expensive. Program verification provides better value if proofs are checked by machine, and preferably generated automatically. The properties checked can range from light-weight control-flow properties to full specification. In order to demonstrate that machine reasoning can improve the quality and cost of artifacts of industrial software engineers, a substantial tool-building effort is required. This workshop brought tool-builders together in order to learn about

- Interfaces between tools (e.g., decision procedures and program verifiers)
- Tool integration platforms
- Case studies that particularly excite the tool aspect

VS-EXPERIMENTS: Workshop on Experiments in Verified Software

Rajeev Joshi (NASA/JPL Laboratory for Reliable Software)
Joseph Kiniry (University College Dublin)

Summary: The scope of the workshop included technical and position papers on all aspects of experiments conducted relating to verified software. The organizers are especially interested in the reflective results of past challenges and ongoing experiments. Such projects include:

- The Mondex Case Study: `vsr.sourceforge.net/mondex.htm`
- The Verified File System: `www.cs.york.ac.uk/circus/mc/abz`
- Medical devices: `www.cas.mcmaster.ca/sqrl/pacemaker.htm`

- Verifying Free andOpen Source Software, e.g., the Apache webserver and the KOA e-voting platform

This workshop was meant to be a *working* workshop. Participants were responsible for formulating action plans, based upon current experiences and best-practices, for tackling the challenges inherent in identifying, defining, promoting, executing, sharing, maintaining, and publishing the results of scientific experiments in verified software.

We would like to thank the following: the keynote speakers and tutors; the authors of all submissions; the members of the Programme Committee (95% of all reviews were received on time!); the workshop organizers and participants; Rick Hehner and his team for the local arrangements for the entire event; Richard Paige for conference and workshop publicity; and last—but not least—our Steering Committee, Jay Misra and Tony Hoare. We are also pleased to acknowledge financial support for VSTTE 2008 from US and UK funding agencies: the US National Science Foundation (NSF) (as part of Grant CNS-0627284) and EPSRC (as part of grant EP/D506735/1), and from Microsoft Research. The proceedings were assembled using EasyChair.

July 2008

Natarajan Shankar
Jim Woodcock

Conference Organization

Steering Committee

Tony Hoare	Microsoft Research Cambridge
Jay Misra	University of Texas at Austin

Programme Chairs

Natarajan Shankar	SRI International
Jim Woodcock	University of York

Programme Committee

Egon Börger	University of Pisa
Supratik Chakraborty	Indian Institute of Technology, Bombay
Patrick Cousot	École Normale Supérieure, Paris
Jin Song Dong	National University of Singapore
José-Luiz Fiadeiro	University of Leicester
Kokichi Futatsugi	JAIST
Chris George	UNU-IIST
Ian Hayes	University of Queensland
Eric Hehner	University of Toronto
Rajeev Joshi	Jet Propulsion Laboratory
Joseph Kiniry	University College Dublin
Yassine Lakhnech	Université Joseph Fourier
Gary Leavens	University of Central Florida
Zhiming Liu	UNU-IIST
Peter Manolios	Northeastern University
Tiziana Margaria	University of Potsdam
David Naumann	Stevens Institute
Peter O'Hearn	Queen Mary, University of London
Ernst-Rüdiger Olderog	University of Oldenburg
Wolfgang Paul	Saarland University
Augusto Sampaio	Federal University of Pernambuco
Mark Utting	Waikato University
Jian Zhang	Chinese Academy of Sciences

Local Organization

Eric Hehner	University of Toronto

Publicity Chair

Richard Paige University of York

External Reviewers

Oliver R. Athing
Stephen Bloom
Ben Chambers
Chunqing Chen
Zhenbang Chen
Yuki Chiba
Antonio Cisternino
Dermot Cochran
Robert Colvin
Márcio Cornélio
Jed Davis
Peter Dillinger
Brijesh Dongol
Fintan Fairmichael
Yuzhang Feng
Sibylle Fröschle
Daniel Gaina
Radu Grigore
Bhargav Gulavani
Yu Guo
Mark Hillebrand
Viliam Holub
Georg Jung
E.-Y. Kang
Ioannis Kassios
Weiqiang Kong
Yang Liu
Charles Morisset
Masaki Nakamura
Zhaozhong Ni
Kazuhiro Ogata
Stan Rosenberg
Andreas Roth
Joseph Ruskiewicz
Gerhard Schellhorn
Norbert Schirmer
Zhong Shao
Leila Silva
Graeme Smith
Volker Stolz
Jun Sun
Aaron Turon
Kapil Vaswani
Xian Zhang

Table of Contents

Keynote Talks (Abstracts)

Logics

Tools

Case Studies

Methodology

Verisoft

Paper from VSTTE 2005

Readable Formal Proofs

John C. Reynolds

Carnegie Mellon University

Abstract. The need to integrate the processes of programming and program verification requires notations for formal proofs that are easily readable. We discuss this problem in the context of Hoare logic and separation logic.

It has long been the custom to describe formal proofs in these logics informally by means of "annotated specifications" or "proof outlines". For simple programs, these annotated specifications are essentially similar to the annotated flow charts introduced by Floyd and Naur. For more elaborate programs, a richer notation has evolved for dealing with procedure calls and various structural rules, such as the frame axiom, as well as various rules for concurrency.

Our goal is to devise a formalism for insuring that annotated specifications actually determine valid formal proofs (modulo the correctness of verification conditions), while providing as much flexibility as possible. For this purpose, we give inference rules for "annotation definitions" that assert that an annotated specification determines a particular Hoare triple. We consider verification algorithms in a wide sense. The outcome of a verification algorithm can be a definite (yes or no) answer, a "don't know" answer, or a conditional answer or no answer at all (divergence). We obtain these kinds of verification algorithms if we apply the existing technology of abstraction to least-fixpoint checking, i.e., checking whether the least fixpoint of a given operator in a given lattice is smaller than a given bound. The formulation of the verification algorithm as least-fixpoint checking is classical for the class of correctness properties that are reducible to non-reachability (validity of assertions, partial correctness, safety properties). We need to investigate the approach also for the class of correctness properties that are reducible to termination (validity of intermittent assertions, total correctness, liveness properties), for all classes of programs including procedural (recursive) programs and concurrent programs.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, p. 1, 2008.

From Verification to Synthesis

Moshe Y. Vardi*

Rice University, Department of Computer Science, Rice University,
Houston, TX 77251-1892, U.S.A.
vardi@cs.rice.edu
http://www.cs.rice.edu/~vardi

Abstract. One of the most significant developments in the area of design verification over the last decade is the development of of algorithmic methods for verifying temporal specification of finite-state designs [2]. A frequent criticism against this approach, however, is that verification is done after significant resources have already been invested in the development of the design. Since designs invariably contains errors, verification simply becomes part of the debugging process. The critics argue that the desired goal ought to be the use of the specification in the design development process in order to guarantee the development of correct designs. This is called design synthesis [1,4]. In this talk I will review 50 years of research on the synthesis problem and show how the automata-theoretic approach can be used to solve it [3,4,5].

References

1. Church, A.: Applicaton of recursive arithmetics to the problem of circuit synthesis. In: Summaries of Talks Presented at The Summer Institute for Symbolic Logic, Communications Research Division, Institute for Defense Analysis, pp. 3–50 (1957)
2. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
3. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science, pp. 531–540 (2005)
4. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages, pp. 179–190 (1989)
5. Rabin, M.O.: Automata on infinite objects and Church's problem. Amer. Mathematical Society (1972)

* Supported in part by NSF grants CCR-0124077, CCR-0311326, CCF-0613889, ANI-0216467, and CCF-0728882, by BSF grant 9800096, and by a gift from Intel.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, p. 2, 2008.

Verification, Least-Fixpoint Checking, Abstraction

Andreas Podelski

University of Freiburg

Abstract. We consider verification algorithms in a wide sense. The outcome of a verification algorithm can be a definite (yes or no) answer, a "don't know" answer, or a conditional answer or no answer at all (divergence). We obtain these kinds of verification algorithms if we apply the existing technology of abstraction to least-fixpoint checking, i.e., checking whether the least fixpoint of a given operator in a given lattice is smaller than a given bound. The formulation of the verification algorithm as least-fixpoint checking is classical for the class of correctness properties that are reducible to non-reachability (validity of assertions, partial correctness, safety properties). We need to investigate the approach also for the class of correctness properties that are reducible to termination (validity of intermittent assertions, total correctness, liveness properties), for all classes of programs including procedural (recursive) programs and concurrent programs.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, p. 3, 2008.

Combining Tests and Proofs

Madhu Gopinathan[1], Aditya Nori[2], and Sriram Rajamani[2]

[1] Indian Institute of Science
madhu@csa.iisc.ernet.in
[2] Microsoft Research India
adityan@microsoft.com, sriram@microsoft.com

Proof methods (or static analysis) and test methods (or dynamic analysis) have complementary strengths. While static analysis has the potential to obtain high coverage, it typically suffers from imprecision (and imprecision is needed to scale the analysis to large programs). While dynamic analysis has the potential to be very precise, it typically suffers from poor coverage.

One approach to combining both is exemplified by the CCured project [1]. Here, the goal is to verify that each memory access done by the program is safe. CCured first attempts to prove that each check is safe statically, using type inference. For the accesses that the static type system is unable to prove safe (due to imprecision), the CCured system instruments checks that are performed at runtime to ensure memory safety.

In this talk, we present two new recipes for combining static and dynamic analysis.

The first recipe is in the context of counter-example driven refinement techniques for proving safety properties of programs. Here, the goal is to check if a program P satisfies a safety property φ, specified as a state machine. We present a counter-example driven refinement technique that combines verification and testing [2]. In our approach, we simultaneously use testing and proving, with the goal of either finding a test that demonstrates that P violates φ, or a proof that demonstrates that all executions of P satisfy φ. The most interesting aspect of the approach is that unsuccessful proof attempts are used to generate tests, and unsuccessful attempts to generate tests are used to refine proofs. Besides being theoretically elegant, the approach has practical advantages —precise alias information obtained during tests can be used to greatly aid the efficiency of constructing proofs [3].

The second recipe is in the context of checking object invariants in object oriented programs. Checking object invariants, even at runtime, is a hard problem. This is because, the object invariant of an object o can depend on another object p and it may not hold when p is modified without o's knowledge. Therefore, whenever an object p is modified, a runtime checker will have to check the object invariant of all objects o that depend on p.

Whenever an object p is modified, a runtime checker will have to check the object invariant for all objects o such that o depends on p. Keeping track of all such dependencies at runtime can slow down a checker significantly. Interestingly, for a large class of object invariants (called object protocol invariants) we can

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 4–5, 2008.

factor out an object invariant in such a way that certain parts of the invariant can be checked statically [4]. The approach has two advantages: (1) a certain class of errors can be detected statically, and (2) the runtime overhead of checking is greatly reduced.

Acknowledgement. We thank our collaborators Nels Beckman, Bhargav Gulavani, Tom Henzinger, Yamini Kannan, Rob Simmons, Sai Tetali and Aditya Thakur.

References

1. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy code. In: POPL 2002: Principles of Programming Languages, pp. 128–139. ACM Press, New York (2002)
2. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: A new algorithm for property checking. In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047. pp. 117–127. Springer, Heidelberg (2006)
3. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA 2008: International Symposium on Software Testing and Analysis. ACM Press, New York (to appear, 2008)
4. Gopinathan, M., Rajamani, S.K.: Enforcing Object Protocols by Combining Static and Dynamic Analysis. In: OOPSLA 2008: ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications. ACM Press, New York (to appear, 2008)

Propositional Dynamic Logic for Recursive Procedures

Daniel Leivant

Computer Science Department, Indiana University, Bloomington, IN 47405, USA
leivant@cs.indiana.edu

Abstract. We present a simple and natural deductive formalism μPDL for propositional dynamic logic for recursive procedures, including simultaneous recursion. Though PDL with recursive programs is known to be highly undecidable, natural deductive formalisms for it are of substantial interest, because they distill the essential logical components of recursive procedures. We also show that Pratt-Kozen's μ-Calculus, in which fixpoints are taken over formulas rather than programs, is interpretable in μPDL.

Keywords: Propositional dynamic logic, recursive procedures, fixpoints, μ-Calculus.

1 Introduction

Logics for programs with recursion have been studied along at least three approaches. One is the semantic and computational complexity study of propositional dynamic logic (PDL) for recursion, as in [7, 9, 11] (see [4, Chapter X] for a survey of earlier works). Another line of research has concerned Hoare-style logics for partial-correctness assertions (PCAs) for recursive programs, as e.g. in [1, 5, 6, 12, 13, 15]. Finally, a synthesis of the two into a first-order dynamic logic for recursive programs was started by Harel in his dissertation [2, 3].

The Pratt-Segerberg formalisms for propositional dynamic logic (PDL) and first-order dynamic logic DL were originally formulated for regular programs, and can be easily modified to apply to guarded iterative (i.e. "while") programs (see [4] for a survey and further references.) DL for guarded iterative programs is an attractive generalization of Hoare's logic, in which one can articulate and prove statements far more general than partial correctness assertions.

An important reason for studying propositional modal logics is to develop practical tools for reasoning and implementation. Decidability, preferably of manageable complexity, is central here. That goal is indeed unattainable for PDL with recursive procedures, since it is highly undecidable, namely Π^1_1-complete, already for rather simple non-regular programs, such as $a^\Delta b a^\Delta$ [7].

This, however, does not diminish the value of the other rationale for studying modal logics, namely distilling the essential components of the topic considered, e.g. time, certainty, knowledge, or the effect of imperative programs. The study of PDL for recursive procedures is a pre-requisite to further study and development of first-order dynamic logic for recursive procedures and other programming constructs and paradigms, and the importance of such deductive calculi is evident, not less so than formalisms for other undecidable structures, such as Peano Arithmetic and Higher Order Logic.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 6–14, 2008.

In this paper we present a simple deductive calculus μPDL for propositional dynamic logic with recursive procedures, and show that Kozen's μ-calculus is interpretable in it.

2 Propositional Dynamic Logic for Recursive Procedures

2.1 PDL for Regular Programs

We recall the essentials of propositional dynamic logic (PDL) for regular programs (for details see e.g. [4]). We posit that *propositional formulas* are generated from a collection of propositional identifiers $p_0, p_1 \ldots$ (for which we use discourse parameters $p, q, \ldots$, possibly decorated), using the connectives $\neg, \wedge, \vee, \rightarrow$. The *atomic programs* of PDL are uninterpreted action-identifiers $a_0, a_1 \ldots$ (for which we use discourse parameters $a, b, \ldots$, possibly decorated), as well as *tests,* i.e. expressions of the form $?\varphi$ where φ is a propositional formula. *Regular programs* are generated from these atomic programs by the operations of union $\cup$, concatenation ;, and iteration $*$. The *PDL formulas* are generated from propositional identifiers $p_0, p_1 \ldots$ by the connectives as well as the modal box $[\cdot]$ and diamond $\langle\cdot\rangle$ operators: for each program α and PDL formula φ, $[\alpha]\varphi$ and $\langle\alpha\rangle\varphi$ are formulas. We often consider one of these modal operators, with the other treated similarly, or — equivalently — defined as the dual of the other, e.g. $\langle\alpha\rangle\varphi \equiv \neg[\alpha]\neg\varphi$.

PDL formulas are interpreted semantically over propositional transition structures. Each such structure $\mathcal{T}$ consists of a set $|\mathcal{T}|$ of *states*, for each propositional identifier p a set $[\![p]\!] \subseteq |\mathcal{T}|$, and for each action-identifier a a relation $[\![a]\!] \subseteq |\mathcal{T}|^2$. The semantic valuation of formulas and programs is obtained by structural recurrence.

PDL for regular programs has the finite-model property, is decidable in exponential time, and is completely axiomatized by the Pratt-Segerberg axioms,

First-order logic		
Modality	Generalization:	$\dfrac{\vdash \varphi}{\vdash [\alpha]\varphi}$
	Distribution:	$[\alpha](\varphi \rightarrow \psi) \rightarrow ([\alpha]\varphi \rightarrow [\alpha]\psi)$
Atomic Programs	Test:	$[?\chi]\varphi \leftrightarrow (\chi \rightarrow \varphi)$
Program constructs	Composition:	$[\alpha;\beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$
	Union:	$[\alpha \cup \beta]\varphi \leftrightarrow [\alpha]\varphi \wedge [\beta]\varphi$
	Iteration:	$[\alpha^*]\varphi \rightarrow \varphi \wedge [\alpha][\alpha^*]\varphi$
	Invariance:	$\dfrac{\vdash \varphi \rightarrow [\alpha]\varphi}{\vdash \varphi \rightarrow [\alpha^*]\varphi}$

From the Invariance Rule we obtain the Schema of Induction:

$$\psi \wedge [\alpha^*](\psi \rightarrow [\alpha]\psi) \rightarrow [\alpha^*]\psi$$

Indeed, taking $\varphi \equiv \psi \wedge [\alpha^*](\psi \rightarrow [\alpha]\psi)$, we have $\vdash \varphi \rightarrow [\alpha]\varphi$ using the remaining axioms and rules, and so $\varphi \rightarrow [\alpha^*]\varphi$ by Iteration. The Induction template above readily follows.

Also, the converse of the Iteration Schema is easily provable: Taking $\psi \equiv \varphi \wedge [\alpha;\alpha^*]\varphi$, we have $\vdash \psi \to [\alpha]\psi$, by Iteration and the modality rules, yielding $\psi \to [\alpha^*]\psi$ by Invariance, while $[\alpha^*]\psi \to [\alpha^*]\varphi$ by Iteration and modality rules.

2.2 PDL for Recursive Programs

An extension of PDL with context-free programs seems to have been proposed first in [5], with striking decidability and undecidability results in subsequent works (see [4, Ch.9] for survey and reference). It is easy to see that the validity problem for PDL augmented with any context-free programs is Π^1_1; surprisingly, the problem is Π^1_1-complete already for certain simple programs, such as $a^\Delta b a^\Delta = \{a^i b a^i \mid i \geqslant 0\}$ [4].

We consider here a syntactically uniform formalism for such extensions of PDL, μPDL, in which a μ operator over programs is used. Although μPDL is Π^1_1-complete [5], it is of great interest because it represents the essence of recursive procedures. We propose a proof system for μPDL, which albeit incomplete (since it generates a Σ^0_1 set of formulas), is natural and of substantial interest: we shall later that its first-order variant is complete in an appropriate sense (analogous to Cook's relative completeness).

Note that this is unrelated to the propositional fixpoint logics of Pratt [14] and Kozen [8], which incorporate fixpoint over propositions, and which are both well-known to be decidable. In contrast, the fixpoints we consider are for recursive *programs,* and as noticed lead to very high undecidability.

The *simple μ-programs* are generated inductively from the action-identifiers and simple tests by three operations: composition, union, and fixpoint (μ). That is, if α and β are programs, then so are $\alpha;\beta$, $\alpha \cup \beta$, and $\mu a.\alpha$, where a is an atomic program-identifier. (As usual, we say that a is *bound* in $\mu a.\alpha$, and more precisely that each occurrence of a in α is bound in $\mu a.\alpha$.) For example, the program $\mu a.(?\top \cup a;\beta)$ is the same, under the intended semantics to be defined momentarily, as β^*. And $\mu c.(?\top \cup a;c;b)$ is the same as the program $a^\Delta b^\Delta = \{a^n b^n \mid n \geqslant 0\}$.

More generally, we consider simultaneous recursion. If $\alpha_1, \ldots, \alpha_m$ are programs, and $a_1, \ldots, a_m$ action-identifiers, then (for $i = 1..m$) $\mu_i a_1 \ldots a_m.(\alpha_1 \ldots \alpha_m)$ is a program. We also consider the latter as the i'th projection of an m-ary vector of programs, $\mu \boldsymbol{a}.\boldsymbol{\alpha} \;=\; \mu a_1 \ldots a_m.(\alpha_1 \ldots \alpha_m)$.[1] Simultaneous recursion can be used to define any context-free program (in the sense of [4]). For example

$$\mu_1 S, A, B.(?\top \cup aB \cup bA,\ aS \cup bAA,\ bS \cup aBB)$$

is the program $P \subseteq \{a,b\}^*$ consisting of traces with an equal number of a's an b's.

A more general formalism is obtained by admitting as tests arbitrary μ-formulas, rather than only pure propositional formulas.[2] That is, one defines by simultaneous structural recurrence both the programs and the formulas. A subtlety is that action-identifiers may then occur in tests in negative position, which should be avoided in the scope of μ. The programs and formulas are thus defined by simultaneous structural recurrence, jointly also with the definition of what it means for an action-identifier a to

[1] Of course, concrete syntax would require parentheses, and the usual precedence conventions would spare the need to display all of them.

[2] Such tests are often dubbed "rich tests", see [4].

occur positively (or negatively) in an expression e (where e is a program or a formula). We use the phrase "a is positive in e" for "a occurs positively in e", and similarly for "negative". Note that a may be both positive and negative in e.

- Action-identifiers a are programs, and a is positive in a. Propositional-identifiers, as well as $\top$ and $\bot$, are formulas. (We will take $\neg\varphi$ to be an abbreviation of $\varphi \to \bot$.)
- If α and β are programs, then so are $\alpha;\beta$ and $\alpha \cup \beta$. If a is positive [negative] in α or β, then it is positive [negative] in $\alpha;\beta$ and $\alpha \cup \beta$.
- If φ, ψ, and χ are formulas, then so are $\psi \wedge \varphi$, $\psi \vee \varphi$ and $\chi \to \varphi$. If a is positive [negative] in φ or ψ, then it is positive [negative] in these formulas; if a is positive [negative] in χ, then it is negative [positive] in $\chi \to \varphi$.
- If φ is a formula, then $?\varphi$ is a program. If a is positive [negative] in φ, then it is positive [negative] in $?\varphi$.
- If α is a program and φ a formula, then $[\alpha]\varphi$ and $\langle\alpha\rangle\varphi$ are formulas. If a is positive [negative] in φ, then it is positive [negative] in these formulas. If a is positive [negative] in α, then it is positive [negative] in $\langle\alpha\rangle\varphi$ and negative [positive] in $[\alpha]\varphi$.
- If α is a program, and a is not negative in α, then $\mu a.\alpha$ is a program. If an action-identifier b, other than a, is positive [negative] in α, then it is positive [negative] in $\mu a.\alpha$. The identifier a itself is of course not free in $\mu a.\alpha$, and thus is neither positive nor negative in this program.

2.3 Semantics of μPDL

The semantics of a program $\mu a.\alpha(a)$ is defined in terms of approximations α^n given by $\alpha^0 = ?\bot$, $\alpha^{n+1} = \alpha(\alpha^n)$. Then $[\![\mu a.\alpha(a)]\!] =_{\mathrm{df}} \cup_n [\![\alpha^n]\!]$. More generally, for a vector $\boldsymbol{\alpha} = \alpha_1 \ldots \alpha_m$ of programs, we define $\alpha_i^0 = ?\bot$, $\alpha_i^{n+1} = \alpha_i(\alpha_1^n \ldots \alpha_m^n)$, and $[\![\mu_i \boldsymbol{a}.\boldsymbol{\alpha}]\!] =_{\mathrm{df}} \cup_n [\![\alpha_i^n]\!]$. (Also, $[\![\mu \boldsymbol{a}.\boldsymbol{\alpha}]\!] =_{\mathrm{df}} (\cup_n [\![\alpha_1^n]\!], \ldots, \cup_n [\![\alpha_m^n]\!])$.) Here $\alpha_i(\alpha_1^n \ldots \alpha_m^n)$ is of course the result of simultaneously replacing in α_i each identifier a_j by the program α_j^n ($j = 1 \ldots m$). The good-behavior of this definition depends on the monotonicity of program semantics, in the following sense.

Proposition 1. *1. If a is positive [negative] in a program α, then for all programs β, γ, if $[\![\beta]\!] \subseteq [\![\gamma]\!]$, then $[\![\alpha(\beta)]\!]$ is a subset [superset] of $[\![\alpha(\gamma)]\!]$.*

2. If a is positive [negative] in a formula φ, then for all programs β, γ, if $[\![\beta]\!] \subseteq [\![\gamma]\!]$, then $[\![\varphi(\beta)]\!]$ is a subset [superset] of $[\![\varphi(\gamma)]\!]$.

Proof. Straightforward (structural) induction on the definition of programs and formulas. ⊣

Corollary 1. *Given a program $\alpha(a)$, we have $[\![\alpha_n]\!] \subseteq [\![\alpha_{n+1}]\!]$ for all n.*

Proof. Straightforward induction on n. ⊣

It follows that the programs α_n behave as approximants from below of $\mu a.\alpha$. Another consequence is that the approximation of $[\![\mu_i \boldsymbol{a}.\boldsymbol{\alpha}]\!] =_{\mathrm{df}} \cup_n [\![\alpha_i^n]\!]$ need not proceed in lockstep, as in the definition above.

2.4 The Expressive Power of μPDL

As noted, μPDL subsumes semantically PDL for regular programs, since for any program β we can define β^* as $\mu a.(?\top \cup (a;\beta))$, where a is a fresh action-identifier. In fact, μPDL subsumes Kozen's μ-Calculus, which itself subsumes PDL for regular programs [8].

Theorem 1. *The μ-calculus is interpretable in μPDL.*

Proof. The proof idea is to represent the propositional fixpoint $\mu p.\varphi$ as a program-fixpoint. Suppose p is positive in $\varphi = \varphi(p)$. Let $\varphi^0 = \bot$, and $\varphi^{n+1} = \varphi(\varphi^n)$. Thus $\mu p.\varphi$ is semantically equivalent to the infinite disjunction $\vee_n \varphi^n$. We use the formula $\langle a\rangle\top$ (a a fresh action-identifier) to represent p, with states where p is true corresponding to states where a is active (i.e. where $\langle a\rangle\top$).

Given a formula $\varphi \equiv \varphi(p)$, let $\alpha(a)$ be the program $?\,\varphi\,(\langle a\rangle\top)$. Let $\alpha^0 = ?\bot$, $\alpha^{n+1} = \alpha(\alpha^n)$. Thus $\mu a.\alpha$ is semantically equivalent to the infinite union $\cup_n \alpha^n$. We prove that the formula φ^n is semantically equivalent to $\langle \alpha^n\rangle\top$, proceeding by induction on n. The case $n = 0$ is immediate. Assuming $\langle \alpha^n\rangle\top \equiv \varphi^n$, we have the semantic equalities

$$\begin{aligned}
\langle \alpha^{n+1}\rangle\top &\equiv \langle\, \alpha(\alpha^n)\,\rangle\top \\
&\equiv \langle\, ?\,\varphi(\langle \alpha^n\rangle\top)\,\rangle\top && \text{(Dfn of } \alpha\text{)} \\
&\equiv \varphi(\langle \alpha^n\rangle\top) && \text{(semantics of tests)} \\
&\equiv \varphi(\varphi^n) && \text{(IH)} \\
&\equiv \varphi^{n+1}
\end{aligned}$$

We thus obtain the semantic equalities

$$\begin{aligned}
\mu p\,\varphi &\equiv \vee_n\, \varphi^n \\
&\equiv \vee_n\, \langle \alpha^n\rangle\top \\
&\equiv \langle \cup \alpha^n\rangle\top \\
&\equiv \langle \mu a.\alpha(a)\rangle\top
\end{aligned}$$

Note that the program $\mu a.(?\,\varphi(\langle a\rangle\top))$ is legal, since p is positive in φ, and a is positive in $\langle a\rangle\top$. ⊣

Of course, Theorem 1 takes nothing off the importance of the μ-Calculus, since the latter is decidable whereas μPDL is highly undecidable:

Theorem 2. [5] *The validity problem for formulas of μPDL is Π^1_1-complete.*

3 Proof Theory of μPDL

3.1 A Deductive Calculus for μPDL

The high undecidability of μPDL does not void the value of deductive calculi for it, just as deductive calculi for Arithmetic and for Higher Order Logic (both highly undecidable) remain of both practical and conceptual interest. Recall that the semantic interpretation $[\![\mu a.\alpha]\!]$ of a recursive program $\mu a.\alpha$ is defined as the fixpoint of

an inductively definition. That is, we posit that $[\![\mu a.\alpha]\!]$ is closed under the operation $\Phi_\alpha : [\![a]\!] \mapsto [\![\alpha(a)]\!]$, i.e. $[\![\mu a.\alpha(a)]\!] \supseteq [\![\alpha(\mu a.\alpha)]\!]$; and that it is the minimal relation with that property. The existence of such a relation can be demonstrated by (ordinal) induction or explicitly as the intersection of all relations closed under Φ_α.

The minimality of $[\![\mu a.\alpha]\!]$ among relations closed under Φ_α is expressed by the informal schema

$$[\![\beta]\!] \supseteq [\![\alpha(\beta)]\!] \;\rightarrow\; [\![\beta]\!] \supseteq [\![\mu a.\alpha]\!] \tag{1}$$

This can be compared to the analogous schema underlying the μ-Calculus, in which one refers to the semantics of formulas, i.e. to sets of states, rather than to programs (binary relations on states):

$$[\![\psi]\!] \supseteq [\![\varphi(\psi)]\!] \;\rightarrow\; [\![\psi]\!] \supseteq [\![\mu p.\varphi]\!] \tag{2}$$

The inclusion in (2) is captured by the semantically sound inference rule:

$$\frac{\vdash \psi(\chi) \rightarrow \chi}{\mu p.\psi \rightarrow \chi}$$

In contrast, the analogous rule for $\mu a.\alpha$, namely

$$\frac{\vdash \langle\alpha(\beta)\rangle\,\varphi \rightarrow \langle\beta\rangle\,\varphi}{\langle\mu a.\alpha\rangle\,\varphi \rightarrow \langle\beta\rangle\,\varphi} \tag{3}$$

is not sound. Indeed, writing $\gamma \upharpoonright \varphi$ for the relation $\{(a,b) \mid a \xrightarrow{\gamma} b \text{ and } b \models \varphi\}$, the premise of (3) states that

$$[\![\beta \upharpoonright \varphi]\!] \;\supseteq\; [\![\alpha(\beta) \upharpoonright \varphi]\!]$$

which indeed implies that

$$[\![\beta \upharpoonright \varphi]\!] \;\supseteq\; [\![\mu a.(\alpha \upharpoonright \varphi)]\!]$$

But the latter does not imply in general

$$[\![\beta \upharpoonright \varphi]\!] \;\supseteq\; [\![(\mu a.\alpha) \upharpoonright \varphi]\!]$$

since $\mu a.(\alpha \upharpoonright \varphi)$ requires φ to hold at the end of every recursive invocation, before proceeding, and not only for the final outcome.

An inference schema that bypasses this snag is the following Minimality natural deduction rule (which we state in a form contra-positive to 3, so as to revert to the use of the box operator):

$$\frac{[\beta]\,p \;\rightarrow\; [\alpha(\beta)]\,p}{[\beta]\,\varphi \;\rightarrow\; [\mu a.\alpha]\,\varphi} \qquad p \text{ not in open assumptions} \tag{4}$$

The schema (4) is semantically sound, because (under the proviso) the premise is equivalent to the second-order formula $\forall p\; \langle\alpha(\beta)\rangle\, p \rightarrow \langle\beta\rangle\, p$, which expresses $[\![\alpha(\beta)]\!] \subseteq [\![\beta]\!]$.

The natural deduction rule (4) does not require the premise to be provable, i.e. open assumptions are permitted (provided p does not occur there). However, the strength of

the premise excludes natural forms of reasoning even about partial correctness assertions. A more useful inference to convey the inclusion (2) is the following *Invariance* natural deduction rule:

$$\dfrac{\begin{array}{c}\{[a]\varphi\}\\ \vdots\\ [\alpha(a)]\varphi\end{array}}{[\mu a.\alpha(a)]\,\varphi} \qquad \begin{array}{l}[a]\varphi \text{ is being closed,}\\ a \text{ not free in } \varphi \text{ or open assumptions}\end{array} \tag{5}$$

Equivalently, and avoiding reference to natural deduction style, Invariance can be formulated as a provability rule:

$$\dfrac{\Gamma,\ [a]\varphi \vdash [\alpha(a)]\varphi}{\Gamma \vdash [\mu a.\alpha(a)]\,\varphi}$$

(where Γ corresponds to the free assumptions of (5)). If Γ is finite, this can be conveyed by

$$\dfrac{\psi \to [a]\varphi \vdash \psi \to [\alpha(a)]\varphi}{\vdash \psi \to [\mu a.\alpha(a)]\,\varphi}$$

where ψ is the conjunction $\wedge\Gamma$. The Invariance Rule (5) is semantically sound, because the premise implies $\Gamma \vdash [\beta]\varphi \to [\alpha(\beta)]\varphi$ for all programs β, in particular for the programs α^n. Thus, by induction on n, we have $\Gamma \models [\alpha^n]\,\varphi$. Since $[\![\mu a.\alpha]\!] = \cup_n [\![\alpha^n]\!]$, we get $\Gamma \models [\mu a.\alpha]\,\varphi$.

Note that the Invariance Rule is similar to Gorelick's familiar Hoare-logic rule for recursive procedures [1].

Returning to the defining properties of $\mu a.\alpha$, the closure of $[\![\mu a.\alpha]\!]$ under Φ_α is conveyed by the following *Unfolding* schema:

$$[\mu a.\alpha(a)]\,\varphi \to [\alpha(\mu a.\alpha)]\,\varphi$$

Natural deduction rules generalizing Minimality, Invariance, and Unfolding to simultaneous recursion are straightforward:

Minimality $\dfrac{\{(\wedge_j\,[\beta_j]\,p_j) \to [\alpha_i(\boldsymbol{\beta})]\,p_i\}_i}{[\beta_j]\,\varphi \to [\mu_j \boldsymbol{a}.\boldsymbol{\alpha}]\,\varphi}$ (no p_i in open assumptions)

Invariance $\dfrac{\{(\wedge_j\,[a_j]\,\varphi_j) \to [\alpha_i(\boldsymbol{a})]\varphi_i\}_i}{[\mu_j \boldsymbol{a}.\boldsymbol{\alpha}(\boldsymbol{a})]\,\varphi_j}$ (no a_i in open assumptions)

Unfolding $[\mu_i \boldsymbol{a}.\boldsymbol{\alpha}]\,\varphi \to [\alpha_i(\mu \boldsymbol{a}.\boldsymbol{\alpha})]\,\varphi$

We define $\boldsymbol{\mu}$**PDL** to be the deductive calculus that consists of the Invariance and Unfolding schemas, applied to simultaneous recursion. The following is straightforward:

Theorem 3. *The deductive calculus* $\boldsymbol{\mu}$**PDL** *is sound.*

Just as first-order theories of inductive definitions are subsumed by second-order logic, the formalism $\boldsymbol{\mu}$**PDL** is subsumed by a theory QPDL that extends PDL with quantifiers over programs [10]. Unfortunately, the leap in complexity is forbidding: the validity problem for QPDL is not even in the analytical hierarchy.

3.2 Interpretation of PDL and the μ-Calculus in μPDL

Theorem 4. *Let φ be a formula of PDL for regular programs, and $\varphi^\sharp$ its translation to μPDL as above. If φ is provable in the Pratt-Segerberg deductive calculus for PDL, then $\varphi^\sharp$ is provable in* $\boldsymbol{\mu}$**PDL**.

Proof. It suffices to show that the Pratt-Segerberg axioms and rules for $*$ are provable in $\boldsymbol{\mu}$**PDL**. To prove the Iteration schema, let us write β for $?\top \cup (a;\alpha)$ and observe that

$$\begin{aligned}
[\alpha^*]\varphi &\equiv [\mu a.\beta]\varphi \\
&\to [\beta(a^*)]\varphi \qquad \text{by Unfolding} \\
&\equiv [?\top \cup a; a^*)]\varphi \\
&\to \varphi \wedge [a;a^*]\varphi \qquad \text{by Union and Test}
\end{aligned}$$

To prove the Invariance schema of PDL for regular program, assume $\vdash \varphi \to [\alpha]\varphi$. Then, for $\beta(a) \equiv (?\top \cup (a;\alpha))$ we have $\vdash [?\top]\varphi \to [\beta(?\top)]\varphi$, and so by the Invariance Rule for recursive programs $[?\top]\varphi \to [\mu a.\beta]\varphi$, i.e. $\varphi \to [a^*]\varphi$. ⊣

More generally, we show that the deductive rules of the μ-Calculus are derivable in $\boldsymbol{\mu}$**PDL**, under the interpretation above.

Theorem 5. *Let φ be a formula of the μ-Calculus, and $\varphi^\sharp$ its translation to μPDL as above. If φ is provable in the μ-Calculus, then $\varphi^\sharp$ is provable in* $\boldsymbol{\mu}$**PDL**.

Proof. Consider an instance $\varphi(\mu p.\varphi) \to \mu p.\varphi$ of the unfolding template of the μ-Calculus. To focus on the essentials assume that φ is μ-free; the general case is treated by structural induction. Let $\alpha_\varphi \equiv \alpha(a) =_{\mathrm{df}} ?\,\varphi(\langle a\rangle\top)$, as above. We have

$$\begin{aligned}
\varphi(\mu p.\varphi)^\sharp &\equiv \varphi(\langle \mu a.\alpha\rangle\top) \\
&\leftrightarrow \langle ?\,\varphi(\langle \mu a.\alpha\rangle\top)\rangle\top \quad \text{(Test Rule)} \\
&\equiv \langle \alpha(\mu a.\alpha)\rangle\top \qquad \text{(Dfn of } \alpha\text{)} \\
&\to \langle \mu a.\alpha\rangle\top \qquad \text{(Unfolding)} \\
&\equiv (\mu p.\varphi)^\sharp
\end{aligned}$$

The remaining rule of the μ-Calculus is

$$\frac{\vdash \varphi(\psi) \to \psi}{\vdash \mu p.\varphi \to \psi}$$

Again, to focus on the essentials we consider the case where φ and ψ are purely propositional. The general case is analogous, using structural induction.

Suppose then that $\varphi(\psi) \to \psi$ is provable in the μ-Calculus (indeed, in pure propositional logic for the case in hand). Thus, in $\boldsymbol{\mu}$**PDL**,

$$\vdash \langle ?\,\varphi(\langle ?\psi\rangle\top)\,\rangle\top \;\to\; \langle ?\psi\rangle\top$$

i.e.

$$\vdash \langle \alpha_\varphi(?\psi)\,\rangle\top \to \langle ?\psi\rangle\top$$

By the contrapositive-form of the Invariance Rule of $\boldsymbol{\mu}$**PDL** this implies

$$\vdash \langle \mu a.\alpha_\varphi \rangle \top \to \psi$$

i.e. exactly

$$\vdash ((\mu p.\varphi) \to \psi)^\sharp$$

$\dashv$

References

1. Gorelick, G.A.: A complete axiomatic system for proving assertions about recursive and nonrecursive programs. Technical Report TR-75, Deptment of Computer Science, University of Toronto (1975)
2. Harel, D.: First-Order Dynamic Logic. LNCS, vol. 68. Springer, Berlin (1979)
3. Harel, D.: Recursion in logics of programs. In: POPL 1979: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 81–92. ACM, New York (1979)
4. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
5. Harel, D., Pnueli, A., Stavi, J.: Propositional dynamic logics of nonregular programs. J. Comput. Sys. Sci. 25, 222–243 (1983)
6. Harel, D., Puneli, A., Stavi, J.: A complete axiomatic system for proving deductions about recursive programs. In: Proceedings of the ninth annual ACM symposium on Theory of computing, pp. 249–260 (1977)
7. Harel, D., Singerman, E.: More on nonregular PDL: finite models and fibonacci-like programs. Inf. Comput. 128(2), 109–118 (1996)
8. Kozen, D.: Results on the propositional mu-calculus. Theoretical Computer Science 27, 333–354 (1983)
9. Lange, M., Somla, R.: Propositional dynamic logic of context-free programs and fixpoint logic with chop. Inf. Process. Lett. 100(2), 72–75 (2006)
10. Leivant, D.: Propositional dynamic logic with program quantifiers. Electronic notes in Theoretical Computer Science (to appear, 2008)
11. Löding, C., Lutz, C., Serre, O.: Propositional dynamic logic with recursive programs. Journal of Logic and Algebraic Programming 73, 51–69 (2007)
12. Nipkow, T.: Hoare logics for recursive procedures and unbounded nondeterminism. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471. pp. 103–119. Springer, Heidelberg (2002)
13. Nipkow, T.: Hoare logics for recursive procedures and unbounded nondeterminism. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471. Springer, Heidelberg (2002)
14. Pratt, V.: A decidable mu-calculus (preliminary report). In: Proceedings of the twenty-second IEEE Symposium on Foundations of Computer Science, pp. 421–427. Computer Society press, Los Angles (1981)
15. von Oheimb, D.: Hoare logic for mutual recursion and local variables. In: Pandu Rangan, C., Raman, V., Ramanujam, R. (eds.) FST TCS 1999. LNCS, vol. 1738. pp. 168–180. Springer, Heidelberg (1999)

Mapped Separation Logic

Rafal Kolanski and Gerwin Klein

Sydney Research Lab., NICTA*, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia
{rafal.kolanski,gerwin.klein}@nicta.com.au

Abstract. We present Mapped Separation Logic, an instance of Separation Logic for reasoning about virtual memory. Our logic is formalised in the Isabelle/HOL theorem prover and it allows reasoning on properties about page tables, direct physical memory access, virtual memory access, and shared memory. Mapped Separation Logic fully supports all rules of abstract Separation Logic, including the frame rule.

1 Introduction

Let memory be a function from addresses to values. Many a paper and semantics lecture start off with this statement. Unfortunately, it is not how memory on any reasonably complex machine behaves. Memory, on modern hardware, is virtualised and addresses pass through a translation layer before physical storage is accessed. Physical storage might be shared between different virtual addresses. Additionally, the encoding of how the translation works (the page table) resides in physical storage as well, and might be accessible through the translation. These two properties make the function model incorrect.

Operating systems can be made to provide the illusion of plain functional memory to applications. This illusion, however, is brittle. Mapping and sharing virtual pages are standard Unix system calls available to applications and they can easily destroy the functional view. What is worse, on the systems software layer, e.g. for the operating system itself, device drivers, or system services, the virtual memory subsystem is fully visible: sharing is used frequently and the translation layer is managed explicitly.

Building on earlier work [11], we show in this paper how Separation Logic [15], a tool for reasoning conveniently about memory and aliasing, can be extended to virtual memory. We do this in a way that preserves the abstract reasoning of Separation Logic as well as its critical locality feature: the frame rule.

The technical contributions in this work are the following:

– A Separation Logic allowing convenient, abstract reasoning about both the virtual and physical layers of memory, supporting the frame rule for arbitrary writes to memory (including the page table),

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 15–29, 2008.

- A separating conjunction extending to virtually shared memory,
- A framework that makes the core logic independent of particular page table implementations, and
- A case study on page allocation that demonstrates the logic.

We currently do not take into account memory permissions beyond whether a virtual address is mapped/unmapped. Such permissions are merely extra properties of virtual addresses; they can be easily integrated into our logic.

All of the work presented in this paper is formalised in the Isabelle/HOL theorem prover [12]. We analyse the logic based on a simple, imperative programming language with arbitrary pointer arithmetic. We embed assertions shallowly and the programming language deeply into Isabelle/HOL. The latter means means we cannot use Isabelle's rich proof automation directly for the case study. The point of this paper is to analyse the logic itself and to prove that it supports the frame rule. The case study shows that it can be applied in principle.

For program verification, we plan to switch to a shallow language embedding and connect this logic with earlier work on a precise memory model of C by Tuch et al. [17], including the automatic verification condition generator used there. As this memory model, the context of the present work is the L4.verified project aiming to prove implementation correctness of the seL4 microkernel [7].

2 Intuition

Separation Logic traditionally models memory as a partial function from addresses to values, called the heap. The two central tools of Separation Logic are the separating conjunction and the frame rule. Separating conjunction $P \wedge^* Q$ is an assertion on heaps, stating that the heap can be split into two separate parts on which the conjuncts P and Q hold respectively. We also say P and Q *consume* disjoint parts of the heap. Separating conjunction conveniently expresses anti-aliasing conditions. For an action f, the frame rule allows us to conclude $\{P \wedge^* R\}\, f\, \{Q \wedge^* R\}$ from $\{P\}\, f\, \{Q\}$ for any R. This expresses that the actions of f are local to the heaps described by P and Q, and can therefore not affect any separate heaps described by R. As we shall see below, with virtual memory, both of these tools break and both can be repaired.

Virtual memory is a hardware-enforced abstraction that allows each executing process its own view of memory, into which areas of physical memory can be inserted dynamically. It adds a level of indirection: virtual addresses potentially map to physical addresses. Memory access is ordinarily done through virtual addresses only, although hardware devices may modify physical memory directly.

This indirection is encoded in the heap itself, in a page table data structure. There are many flavours of such encodings, using multiple levels, different sizes of basic blocks (e.g. super pages), dynamically different table depths, and even hash tables. Usually, the encoding and minimum granularity (page size) is dictated by the hardware. The anchor of the page table in physical memory is called its root. Knowing the specific encoding, the full set of virtual-to-physical mappings for a process can be constructed from the heap and the root.

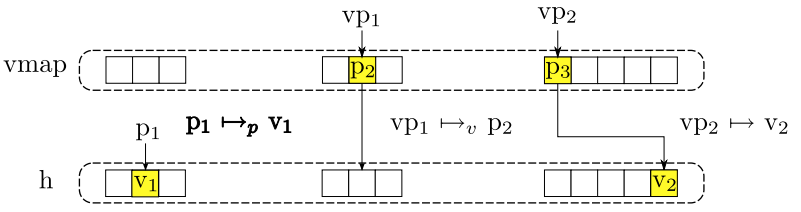


Fig. 1. Maps-to assertions on the heap, virtual map and address space

Given the additional layer of indirection, two different virtual addresses may resolve to the same physical address, and separating conjunction breaks, because on these addresses it does not end up providing separation. Additionally, although a memory update to the page table may only locally change one byte of physical memory, it might completely change the view the virtual memory layer provides, affecting a whole number of seemingly unrelated virtual addresses. A local action might therefore have non-local effects. This breaks the frame rule. We show how to repair both tools for virtual memory in the remainder of this paper.

We will henceforth restrict our use of the word *heap* to refer to physical memory, and we will call the virtual-to-physical translation the *virtual map*. This virtual map provides the first abstraction layer in our model of virtual memory: for the development of Separation Logic on virtual memory, we do not need to know how the encoding works precisely. We only need to know that there exists a way of decoding, and, as we shall see later, that there is a way of finding out which page table entries contribute to looking up a particular virtual address. As the rest of the development is independent of the encoding, we use only a very simple implementation for the case study presented here: a one-level page table.

For reasoning about virtual memory we wish to make assertions on three levels, shown in Fig. 1: physical-to-value, virtual-to-physical, and virtual-to-value. Given our two partial functions (heap and vmap), this may appear straightforward. The virtual-to-physical assertion corresponds to looking up a virtual address in the page table. The central question that arises is: *which part of the heap should a virtual-to-physical assertion consume?* This question determines the meaning of separation between this and other assertions. Assuming for example a one-level page table where memory location x encodes the lookup of virtual address vp_1 to physical address p_2, there are two cases: (a) the lookup consumes x, or (b) it does not. In case (a) we cannot use separating conjunction to specify separation of vp_1 and another address vp_2 if vp_2 happens to use the same page table entry x. Typically, many virtual addresses share an entry in the encoding, and we clearly do not want to exclude this case. The other extreme, in case (b), would be to say the lookup consumes no resources. In earlier work [11] we came to the result that this model cannot support the frame rule for arbitrary programs: a physical write to the page table would be separate to any virtual-to-physical lookup, but the write might have the non-local effect of changing that apparently separate mapping.

In this paper, we are therefore proposing a solution in between: a page table lookup consumes only *parts* of the page table entry involved. This can be seen as cutting each heap location into multiple slices. If a page table lookup, i.e. a virtual-to-physical mapping, consumes parts of the page table entry involved in the lookup, we can make all memory updates local.

This idea is similar to the model of permissions by Bornat et al. [3] for concurrent threads. The difference is that in our case we extend the domain of the heap to slices, whereas in the permission model, the range is extended. This has the advantage that the development of heap disjointness, merging, separating conjunction and the other standard operators remains unchanged, and therefore should be easy to instantiate to our target model of C [17]. The memory footprint of page table lookups has a direct formulation in our model.

The next section will briefly summarise our formal abstract interface to page table encodings, before Sect. 4 introduces the model and shallow Isabelle/HOL embedding of the assertion language for virtual memory.

3 The Virtual Memory Environment

In this section, we describe the page table abstraction our logic is based on, as well as an instantiation to a simple, one-level page table.

As in previous work [11], we write VPtr *vp* and PPtr *p* to distinguish virtual and physical pointers by type in Isabelle/HOL, and provide a destructor ptr-val (PPtr x) $= x$ when we need to talk about the physical/virtual address directly.

In Isabelle the abstract concepts of heap, virtual map, and address space are:

types $heap = pptr \rightharpoonup val$ $\quad addr\text{-}space = vptr \rightharpoonup val$
$vmap = vptr \rightharpoonup pptr$

"$\rightharpoonup$" denotes a partial function. These types do not include slicing yet. Pointer sizes are parameters of the development. We use 32-bit machine words for pointers as well as values of memory cells in concrete examples.

For the development of the logic in the remainder of the paper, we require the following two functions.

ptable-lift :: $heap \Rightarrow pptr \Rightarrow vmap$
ptable-trace :: $heap \Rightarrow pptr \Rightarrow vptr \Rightarrow pptr$ set

Both functions take the physical heap and page table root as parameters. The result of ptable-lift is the *vmap* encoded by the page table, whereas ptable-trace returns for each virtual address *vp* the set of locations in the page table that are involved in the lookup of *vp*. In contrast to our earlier work, we do not require an explicit formulation of the page table area in memory.

We will later require five constraints on these two functions. We defer the presentation of these to Sect. 5 when the associated concepts have been introduced.

As mentioned above, we use a simple one-level page table as an example instantiation. It is a contiguous physical memory structure consisting of an array of machine word pointers, where word 0 defines the physical location of page 0 in the address space, word 1 that of page 1 and so forth. While inefficient in terms

of storage, it is simple to present and experiment with. The table is based on an arbitrarily chosen page size of 4096, i.e. 20 bits for the page number and 12 for the offset. Page table lookup works as expected: we extract the page number from the virtual address, go to that offset in the page table and obtain a physical frame number which replaces the top 20 bits of the address:

$$
\begin{array}{lcl}
\mathsf{get\text{-}page}\ vp & \equiv & \mathsf{ptr\text{-}val}\ vp >> 12 \\
\mathsf{ptr\text{-}remap}\ (\mathsf{VPtr}\ vp)\ pg & \equiv & \mathsf{PPtr}\ (pg\ \mathsf{AND}\ \mathsf{NOT}\ 0xFFF\ \mathsf{OR}\ vp\ \mathsf{AND}\ 0xFFF) \\
\mathsf{ptable\text{-}lift}\ h\ r\ vp & \equiv & \mathsf{case}\ h\ (r + \mathsf{get\text{-}page}\ vp)\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{None} \\
 & & \quad |\ \lfloor addr \rfloor \Rightarrow \\
 & & \quad\quad \mathsf{if}\ addr\ !!\ 0\ \mathsf{then}\ \lfloor \mathsf{ptr\text{-}remap}\ vp\ addr \rfloor\ \mathsf{else}\ \mathsf{None} \\
\mathsf{ptable\text{-}trace}\ h\ r\ vp & \equiv & \mathsf{case}\ h\ (r + \mathsf{get\text{-}page}\ vp)\ \mathsf{of}\ \mathsf{None} \Rightarrow \emptyset \\
 & & \quad |\ \lfloor addr \rfloor \Rightarrow \{r + \mathsf{get\text{-}page}\ vp\}
\end{array}
$$

AND, OR and NOT are bitwise operations on words. The operator $>>$ is bitwise right-shift on words. The term $x\ !!\ n$ stands for bit n in word x. We use bit 0 to denote whether a mapping is valid. The optional value type has two constructors: None for no value and $\lfloor value \rfloor$ otherwise.

4 Separation Logic Assertions on Virtual Memory

In this section, we describe how the classical Separation Logic assertions can be extended to hold for a model with virtual memory.

As mentioned in Sect. 2, the idea is to extend the domain of the heap to slices. One page table entry can, at most, be responsible for all virtual addresses in the machine. Therefore, the smallest useful granularity of slicing up one physical address is that of one slice per virtual address. For each physical address p, we need to be able to address the slice of p responsible for virtual address 0, 1, ... :: *vptr* etc. Thus, the domain of the heap becomes $pptr \times vptr$ and $(p,\ vp)$ stands for the vp-slice of physical address p.

In our shallow embedding, Separation Logic assertions are predicates on this new heap and the root of the page table:

$$
\begin{array}{ll}
\textbf{types} & fheap = (pptr \times vptr) \rightharpoonup val \\
 & map\text{-}assert = (fheap \times pptr) \Rightarrow bool
\end{array}
$$

With this model, the basic definition of heap merge ($++$), domain, and disjointness ($\perp$) remain completely standard:

$$
\begin{array}{lcl}
h_1 ++ h_2 & \equiv & \lambda x.\ \mathsf{case}\ h_2\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow h_1\ x\ |\ \lfloor y \rfloor \Rightarrow \lfloor y \rfloor \\
\mathsf{dom}\ h & \equiv & \{x\ |\ h\ x \neq \mathsf{None}\} \\
h_1 \perp h_2 & \equiv & \mathsf{dom}\ h_1 \cap \mathsf{dom}\ h_2 = \emptyset
\end{array}
$$

The Separation Logic connectives for empty heap, true, conjunction, and implication stay almost unchanged. We additionally supply the page table root r.

$$
\begin{array}{lcl}
\square & \equiv & \lambda(h,\ r).\ h = \mathsf{empty} \\
\top & \equiv & \lambda(h,\ r).\ \mathsf{True} \\
P \wedge^* Q & \equiv & \lambda(h,\ r).\ \exists h_0\ h_1.\ h_0 \perp h_1 \wedge h = h_0 ++ h_1 \wedge P\ (h_0,\ r) \wedge Q\ (h_1,\ r) \\
P \longrightarrow^* Q & \equiv & \lambda(h,\ r).\ \forall h'.\ h \perp h' \wedge P\ (h',\ r) \longrightarrow Q\ (h ++ h',\ r)
\end{array}
$$

Since the definitions are almost unchanged it is unsurprising that the usual properties such as commutativity and associativity and distribution over lifted normal conjunction continue to hold.

The interesting, new assertions are the three *maps-to* statements that we alluded to in Fig. 1. Corresponding to the traditional Separation Logic predicate is the physical-to-value mapping:

$$p \mapsto_p v \equiv \lambda(h,\, r).\ (\forall\, vp.\ h\ (p,\, vp) = \lfloor v \rfloor) \wedge \mathsf{dom}\ h = \{p\} \times \mathcal{U}$$

For this assertion, we require that all slices of the heap at physical address p map to the value v and that the domain of the heap is exactly the set of all pairs with p as the first component ($\mathcal{U}$ is the universe set). This predicate would typically be used for direct memory access in devices or for low-level kernel operations.

The next assertion is the virtual-to-physical mapping:

$$\begin{aligned} vp \mapsto_v p \equiv \lambda(h,\, r).&\\ &\mathsf{let}\ heap = \mathsf{h\text{-}view}\ h\ vp;\ vmap = \mathsf{ptable\text{-}lift}\ heap\ r\\ &\mathsf{in}\ vmap\ vp = \lfloor p \rfloor \wedge \mathsf{dom}\ h = \mathsf{ptable\text{-}trace}\ heap\ r\ vp \times \{vp\} \end{aligned}$$

where $\mathsf{h\text{-}view}\ fh\ vp \equiv \lambda p.\ fh\ (p,\, vp)$. Here, we first lift the page table out of the heap to get the abstract *vmap* which provides us with the translation from *vp* to *p*. Additionally, we assert that the domain of the heap is the *vp* slice of all page table entries that are involved in the lookup. With h-view we project out the *vp* slice for all addresses so that the page table lift function can work on a traditional $pptr \rightharpoonup val$ heap.

Putting the two together, we arrive at the virtual-to-value mapping that corresponds to the level that most of the OS and user code will be reasoning at.

$$vp \mapsto v \equiv \lfloor \exists \rfloor p.\ vp \mapsto_v p \wedge^* p \mapsto_p v$$

where $\lfloor \exists \rfloor x.\ P\ x \equiv \lambda s.\ \exists x.\ P\ x\ s$. The predicate implies that the lookup path is separate from the physical address p the value v is at. This is the case for all situations we have encountered so far and, as the case study shows, works for page table manipulations as well. It is possible to define a weaker predicate without the separation, but this creates a special case for the assignment rule: if a write changes the page table for the address the write goes to, the post condition would have to take the change in the translation layer into account directly.

The usual variations on the maps-to predicate can be defined in the standard manner again for virtual-to-value mappings:

$$\begin{aligned} &p \mapsto - \equiv \lfloor \exists \rfloor v.\ p \mapsto v\\ &p \hookrightarrow v \equiv p \mapsto v \wedge^* \top\\ &S\ \{\mapsto\}\ - \equiv \mathsf{fold\ op} \wedge^*\ (\lambda x.\ x \mapsto -)\ \square\ S \end{aligned}$$

The latter definition refers to a finite set S of addresses. It states that all of the elements map separately by folding the $\wedge^*$ operator over S and the maps-to predicate. There are analogous variations for physical-to-value and virtual-to-physical mappings.

We have proved that our basic mappings $\mapsto_p$ and $\mapsto_v$ are domain exact [15]. Note that, although not domain exact due to the existential quantifier on p, the virtual-to-value mapping is still precise [13]:

$$\mathsf{precise}\ P \equiv \forall\, Q\ R.\ (P \wedge^* Q) \lfloor\wedge\rfloor R) = ((P \wedge^* Q) \lfloor\wedge\rfloor (P \wedge^* R))$$

That is, it distributes over conjunction in both directions. We write $P \lfloor\wedge\rfloor Q$ for the lifted conjunction of assertions P and Q: $P \lfloor\wedge\rfloor Q \equiv \lambda x.\ P\ x \wedge Q\ x$.

5 The Logic

This section introduces a simple, heap based programming language with pointer arithmetic to analyse how the assertions presented above can be developed into a full Separation Logic. For the meta-level proofs in this paper, we provide a deep embedding of the language into Isabelle/HOL. The language is standard, with skip, if, while, and assignment. The WHILE and IF statements potentially read from virtual memory in their guards. Assignment is the most interesting: it accesses memory through the virtual memory layer and can potentially modify this translation by writing to the page table. We leave out the simpler physical write access for the presentation here. It would be easy to add and does not increase the complexity of the language.

Fig. 2 shows the Isabelle datatypes that make up the syntax of the language. Note that the left hand side of assignments can be an arbitrary arithmetic expression. For simplicity, we identify values and pointers in this language and admit arbitrary HOL functions for comparison and arithmetic expressions. The program states are the same states that the separation assertions work on. To keep the number of statements small, we only provide the more complicated case of virtual access: even low-level page table manipulations are translated.

The semantics of boolean and arithmetic expressions is shown in Fig. 3. We write $[\![B]\!]_b$ for the meaning of boolean expression B as a partial function from program state to *bool*. Analogously $[\![A]\!]$ is a partial function from state to values. All of these are straightforward, only heap lookup deserves more attention.

Heap lookup only succeeds if the address the argument of the lookup evaluates to virtually maps to a value. This means that the appropriate slices that are involved in the page table lookup as well as the full cell of the target in physical memory must be available in the domain of the heap. In any execution, there will

datatype *aexp* =
  HeapLookup *aexp*
| BinOp (*val* ⇒ *val* ⇒ *val*) *aexp aexp*
| UnOp (*val* ⇒ *val*) *aexp*
| Const *val*

datatype *com* =
  SKIP
| *aexp* := *aexp*
| *com*; *com*
| IF *bexp* THEN *com* ELSE *com*
| WHILE *bexp* DO *com*

datatype *bexp* =
  BConst *bool*
| BComp (*val* ⇒ *val* ⇒ *bool*) *aexp aexp*
| BBinOp (*bool* ⇒ *bool* ⇒ *bool*) *bexp bexp*
| BNot *bexp*

Fig. 2. Syntax of the heap based WHILE language

$$
\begin{aligned}
&[\![\mathsf{Const}\ c]\!]\ s &&= \lfloor c \rfloor \\
&[\![\mathsf{HeapLookup}\ vp]\!]\ s &&= \mathsf{case}\ [\![vp]\!]\ s\ \mathsf{of\ None} \Rightarrow \mathsf{None} \\
&&&\quad \mid \lfloor v \rfloor \Rightarrow \mathsf{if\ (VPtr}\ v \hookrightarrow -)\ s\ \mathsf{then\ as\text{-}view\ ptable\text{-}lift}\ s\ \mathsf{(VPtr}\ v)\ \mathsf{else\ None} \\
&[\![\mathsf{BinOp}\ f\ e_1\ e_2]\!]\ s &&= \mathsf{case}\ ([\![e_1]\!]\ s,\ [\![e_2]\!]\ s)\ \mathsf{of} \\
&&&\quad (\lfloor v_1 \rfloor,\ \lfloor v_2 \rfloor) \Rightarrow \lfloor f\ v_1\ v_2 \rfloor \mid \text{-} \Rightarrow \mathsf{None} \\
&[\![\mathsf{UnOp}\ f\ e]\!]\ s &&= \mathsf{case}\ [\![e]\!]\ s\ \mathsf{of\ None} \Rightarrow \mathsf{None} \mid \lfloor v \rfloor \Rightarrow \lfloor f\ v \rfloor \\
&[\![\mathsf{BConst}\ b]\!]_b\ s &&= \lfloor b \rfloor \\
&[\![\mathsf{BComp}\ f\ e_1\ e_2]\!]_b\ s &&= \mathsf{case}\ ([\![e_1]\!]\ s,\ [\![e_2]\!]\ s)\ \mathsf{of} \\
&&&\quad (\lfloor v_1 \rfloor,\ \lfloor v_2 \rfloor) \Rightarrow \lfloor f\ v_1\ v_2 \rfloor \mid \text{-} \Rightarrow \mathsf{None} \\
&[\![\mathsf{BBinOp}\ f\ e_1\ e_2]\!]_b\ s &&= \mathsf{case}\ ([\![e_1]\!]_b\ s,\ [\![e_2]\!]_b\ s)\ \mathsf{of} \\
&&&\quad (\lfloor v_1 \rfloor,\ \lfloor v_2 \rfloor) \Rightarrow \lfloor f\ v_1\ v_2 \rfloor \mid \text{-} \Rightarrow \mathsf{None} \\
&[\![\mathsf{BNot}\ b]\!]_b\ s &&= \mathsf{case}\ [\![b]\!]_b\ s\ \mathsf{of\ None} \Rightarrow \mathsf{None} \mid \lfloor v \rfloor \Rightarrow \lfloor \neg\ v \rfloor
\end{aligned}
$$

Fig. 3. Semantics of arithmetic and boolean expressions

$$
\langle \mathsf{SKIP}, s\rangle \rightarrow \lfloor s \rfloor
\qquad
\frac{[\![lval]\!]\ s = \lfloor vp \rfloor \qquad [\![rval]\!]\ s = \lfloor v \rfloor \qquad (\mathsf{VPtr}\ vp \hookrightarrow -)\ s}{\langle lval := rval\ , s\rangle \rightarrow \lfloor s\ [\mathsf{VPtr}\ vp, - \mapsto_v v] \rfloor}
$$

$$
\frac{[\![lval]\!]\ s = \mathsf{None} \vee [\![rval]\!]\ s = \mathsf{None}}{\langle lval := rval\ , s\rangle \rightarrow \mathsf{None}}
\qquad
\frac{[\![lval]\!]\ s = \lfloor vp \rfloor \qquad \neg\ (\mathsf{VPtr}\ vp \hookrightarrow -)\ s}{\langle lval := rval\ , s\rangle \rightarrow \mathsf{None}}
$$

$$
\frac{\langle c_0, s\rangle \rightarrow \lfloor s'' \rfloor \qquad \langle c_1, s''\rangle \rightarrow s'}{\langle c_0\ ;\ c_1, s\rangle \rightarrow s'}
\qquad
\frac{\langle c_0, s\rangle \rightarrow \mathsf{None}}{\langle c_0\ ;\ c_1, s\rangle \rightarrow \mathsf{None}}
$$

$$
\frac{<b> s \qquad \langle c_0, s\rangle \rightarrow s'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1, s\rangle \rightarrow s'}
\qquad
\frac{\neg <b> s \qquad \langle c_1, s\rangle \rightarrow s'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1, s\rangle \rightarrow s'}
$$

$$
\frac{[\![b]\!]_b\ s = \mathsf{None}}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1, s\rangle \rightarrow \mathsf{None}}
\qquad
\frac{\neg <b> s}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \rightarrow \lfloor s \rfloor}
$$

$$
\frac{<b> s \qquad \langle c, s\rangle \rightarrow \lfloor s'' \rfloor \qquad \langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s''\rangle \rightarrow s'}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \rightarrow s'}
$$

$$
\frac{<b> s \qquad \langle c, s\rangle \rightarrow \mathsf{None}}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \rightarrow \mathsf{None}}
\qquad
\frac{[\![b]\!]_b\ s = \mathsf{None}}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \rightarrow \mathsf{None}}
$$

where

$$
\begin{aligned}
&(h,\ r)\ [vp, - \mapsto_v v] &&\equiv \mathsf{case\ vmap\text{-}view}\ (h,\ r)\ vp\ \mathsf{of}\ \lfloor p \rfloor \Rightarrow (h\ [p, - \mapsto v],\ r) \\
&\mathsf{vmap\text{-}view}\ (h,\ r)\ vp &&\equiv \mathsf{ptable\text{-}lift}\ (\mathsf{h\text{-}view}\ h\ vp)\ r\ vp \\
&h\ [p, - \mapsto v] &&\equiv \lambda p'.\ \mathsf{if\ fst}\ p' = p\ \mathsf{then}\ \lfloor v \rfloor\ \mathsf{else}\ h\ p'
\end{aligned}
$$

Fig. 4. Big-step semantics of commands

always be full memory cells available, but using our assertions from above we are able to make finer-grained distinctions during program proofs. The function $\mathsf{as\text{-}view\ ptable\text{-}lift}\ (fh,\ r)\ vp = (\mathsf{let}\ hp = \mathsf{h\text{-}view}\ fh\ vp\ \mathsf{in\ ptable\text{-}lift}\ hp\ r \rhd hp)\ vp$ uses h-view to read the value from the heap after the address has been translated, and $f \rhd g \equiv \lambda x.\ \mathsf{case}\ f\ x\ \mathsf{of\ None} \Rightarrow \mathsf{None} \mid \lfloor y \rfloor \Rightarrow g\ y$ to compose the two partial functions.

Finally, Fig. 4 shows a big-step operational semantics of commands in the language. We write $\langle c, s\rangle \rightarrow s'$ if command c, started in s, evaluates to s'. As usual, the semantics is the smallest relation satisfying the rules of Fig. 4. Non-termination

is modelled by absence of the transition from the relation. In contrast to this, we model memory access failure explicitly by writing $\langle c,s\rangle \rightarrow \mathsf{None}$, and we do not allow accessing unmapped pages. On hardware, this would lead to a page fault and the execution of a page fault handler. This can also be modelled by making the assignment statement a conditional jump to the page fault handler, or with an abstraction layer showing that the page fault handler always establishes a known good state by mapping pages in from disk. For the verification of the seL4 microkernel [6,7] (the larger context of this work), we intend to show absence of page faults in the kernel itself, hence our stricter model. Excluding the conditional jump is no loss of generality as we already have IF statements in the language.

As with arithmetic expressions, the most interesting rule in Fig. 4 is assignment which involves heap access and which we will explain below. In the other rules, we abbreviate $[\![b]\!]_b\ s = \lfloor \mathsf{True} \rfloor$ with $<b>\ s$, and $[\![b]\!]_b\ s = \lfloor \mathsf{False} \rfloor$ with $\neg<b>\ s$. Failure in any part of the execution leads to failure of the whole statement.

The assignment rule in the top right corner of Fig. 4 requires that both the arithmetic expressions for the left and right hand side evaluate without failure. The left hand side is taken as a virtual pointer, the right hand side as the value being assigned. The assignment succeeds if the virtual address vp is mapped and allocated. We use the notation $s\ [vp,- \mapsto v]$ to update with v all slices in the heap belonging to the physical address that vp resolves to. Since heap writes always update such full cells, all heaps in executions will only ever consist of full cells and are thus consistent with the usual, non-sliced view of memory. Slices are a tool for assertions and proofs only.

Having shown the semantics, we can now proceed to defining Hoare triples. Validity is the usual:

$$\{\!|P|\!\}\ c\ \{\!|Q|\!\} \equiv \forall\, s\ s'.\ \langle c,s\rangle \rightarrow s' \wedge P\ s \longrightarrow (\exists\, r.\ s' = \lfloor r \rfloor \wedge Q\ r)$$

We do not define a separate syntactic Hoare calculus. Instead, we define validity only and then derive the Hoare rules as theorems in Isabelle/HOL directly. Fig. 5 shows the rules we have proved for this language. Again, the rules for IF, WHILE, etc., are straightforward and the same as in a standard Hoare calculus. We write $P\ \lfloor\!\longrightarrow\!\rfloor\ Q \equiv \forall\, s.\ P\ s \longrightarrow Q\ s$ for lifted implication, and $\ll\!b\!\gg\ s$ to denote that $[\![b]\!]_b\ s \neq \mathsf{None}$. The precondition P in the IF and WHILE case must be strong enough to guarantee failure free evaluation of the condition b. The lifting rules for conjunction and disjunction, as well as the weakening rule are easy to prove, requiring only the definition of validity and separating conjunction. The interesting cases are the assignment rule and the frame rule.

The assignment rule is more complex looking than the standard rule of Separation Logic. Here, both the left and right hand side of the assignment are arbitrary expressions, potentially including heap lookups that need to be evaluated first. This is the reason for the additional P conjunct separate from the basic pointer mapping. It is not an artifact of virtual memory, but one of expression evaluation only. In essence, this is a rule schema. P can be picked to be just strong enough for the evaluation of the expressions to succeed. For instance, if the left hand side were to contain one heap lookup for location x only, we would choose P to be of the form $x \mapsto r$. The rule is sound for too strong and too weak

$$\{\!|P|\!\}\ \mathsf{SKIP}\ \{\!|P|\!\} \qquad \frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \qquad P' \lfloor\longrightarrow\rfloor P \qquad Q \lfloor\longrightarrow\rfloor Q'}{\{\!|P'|\!\}\ c\ \{\!|Q'|\!\}}$$

$$\frac{\{\!|P \lfloor\wedge\rfloor <b>|\!\}\ c_1\ \{\!|Q|\!\} \qquad \{\!|P \lfloor\wedge\rfloor \neg<b>|\!\}\ c_2\ \{\!|Q|\!\}}{\{\!|P \lfloor\wedge\rfloor \ll b \gg|\!\}\ \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2\ \{\!|Q|\!\}}$$

$$\frac{\{\!|P \lfloor\wedge\rfloor <b>|\!\}\ c\ \{\!|P|\!\} \qquad P \lfloor\longrightarrow\rfloor \ll b \gg}{\{\!|P|\!\}\ \mathsf{WHILE}\ b\ \mathsf{DO}\ c\ \{\!|P \lfloor\wedge\rfloor \neg<b>|\!\}} \qquad \frac{\{\!|P|\!\}\ c_1\ \{\!|Q|\!\} \qquad \{\!|Q|\!\}\ c_2\ \{\!|R|\!\}}{\{\!|P|\!\}\ c_1\ ;\ c_2\ \{\!|R|\!\}}$$

$$\frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \qquad \{\!|R|\!\}\ c\ \{\!|S|\!\}}{\{\!|P \lfloor\wedge\rfloor R|\!\}\ c\ \{\!|Q \lfloor\wedge\rfloor S|\!\}} \qquad \frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \qquad \{\!|R|\!\}\ c\ \{\!|S|\!\}}{\{\!|P \lfloor\vee\rfloor R|\!\}\ c\ \{\!|Q \lfloor\vee\rfloor S|\!\}}$$

$$\{\!|(\mathsf{VPtr}\ vp \mapsto - \wedge^*\ P) \lfloor\wedge\rfloor [\![l]\!] = \lfloor vp \rfloor \lfloor\wedge\rfloor [\![r]\!] = \lfloor v \rfloor|\!\}\ l := r\ \{\!|\mathsf{VPtr}\ vp \mapsto v \wedge^*\ P|\!\}$$

$$\frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\}}{\{\!|P \wedge^*\ R|\!\}\ c\ \{\!|Q \wedge^*\ R|\!\}}$$

Fig. 5. The proof rules for Mapped Separation Logic

P: either the precondition is merely stronger than it needs to be, or P is too weak to support the expression evaluation conjunct and the precondition as a whole becomes false.

The proof of the assignment rule proceeds by unfolding the definitions and observing that the postcondition is established by the memory update, noting the fact that the ptable-trace from the precondition fully accounts for all page table entries that are relevant in the postcondition and that P is preserved, because it is separate from the heap in which the update occurs. Since the physical address of the write is separate from the page table lookup for this address and from P, the translation layer for their heaps is not affected by the write. To reason about page table updates we need a slightly stronger rule that unfolds the virtual-to-value mapping and lets us talk about the physical address p:

$$\begin{array}{c}\{\!|(\mathsf{VPtr}\ vp \mapsto_v p \wedge^*\ p \rightarrow_p - \wedge^*\ P) \lfloor\wedge\rfloor [\![l]\!] = \lfloor vp \rfloor \lfloor\wedge\rfloor [\![r]\!] = \lfloor v \rfloor|\!\} \\ l := r \\ \{\!|\mathsf{VPtr}\ vp \mapsto_v p \wedge^*\ p \mapsto_p v \wedge^*\ P|\!\}\end{array}$$

Reasoning with the new mapping predicates is similar to abstract-predicate style reasoning [14] if we never unfold their definitions in client proofs. As we will see in the case study, we only need to do this locally if we are reasoning about changes to the page table and we are interested in the page that is being modified. This obviously heavily depends on the page table encoding. Application level reasoning can proceed fully abstractly.

The second interesting proof rule is the frame rule that allows global reasoning based on local proofs. In many ways it can be seen as the core of Separation Logic. Calcagno et al. [4] provide an abstract framework for identifying a logic as Separation Logic and distill out the central property of locality. In their setting,

$$\frac{\mathsf{ptable\text{-}lift}\ (h_0 +\!\!+ h_1)\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \lfloor p \rfloor \vee \mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \mathsf{None}} \qquad \frac{\mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\mathsf{ptable\text{-}lift}\ (h_0 +\!\!+ h_1)\ r\ vp = \lfloor p \rfloor}$$

$$\frac{\mathsf{get\text{-}page}\ vp = \mathsf{get\text{-}page}\ vp' \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor val \rfloor \qquad \mathsf{ptable\text{-}lift}\ h'\ r\ vp' = \lfloor val' \rfloor}{\mathsf{ptable\text{-}trace}\ h\ r\ vp = \mathsf{ptable\text{-}trace}\ h'\ r\ vp'}$$

$$\frac{p \notin \mathsf{ptable\text{-}trace}\ h\ r\ vp \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor p \rfloor}{\mathsf{ptable\text{-}lift}\ (h(p \mapsto v))\ r\ vp = \lfloor p \rfloor}$$

$$\frac{p \notin \mathsf{ptable\text{-}trace}\ h\ r\ vp \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor p \rfloor}{\mathsf{ptable\text{-}trace}\ (h(p \mapsto v))\ r\ vp = \mathsf{ptable\text{-}trace}\ h\ r\ vp}$$

Fig. 6. The page table interface

our separation algebra is the common heap monoid where the binary operation is heap merge lifted to the *heap* × *pptr* type. Our definition of separating conjunction then coincides with the one in the framework, and to show that our logic is a Separation Logic, we only need to show that all actions in the programming language are local. Locality is equivalent to the combination of safety monotonicity and the frame property [4]. In our setting, these two are:

Lemma 1 (Safety Monotonicity). *If a small state provides enough resources to run a command c, then so does a larger state. Formally, the converse is easier to state: If* $\langle c,(h +\!\!+ h', r)\rangle \to \mathsf{None}$ *and* $h \perp h'$ *then* $\langle c,(h, r)\rangle \to \mathsf{None}$.

Lemma 2 (Frame Monotonicity). *Execution of a command can be traced back to a smaller part of the state as long as the command executes at all on the smaller state. If* $\neg\ \langle c,(h_0, r)\rangle \to \mathsf{None}$ *and* $\langle c,(h_0 +\!\!+ h_1, r)\rangle \to \lfloor (h', r)\rfloor$ *and* $h_0 \perp h_1$ *then* $\exists\, h_0'.\ h' = h_0' +\!\!+ h_1 \wedge \langle c,(h_0, r)\rangle \to \lfloor (h_0', r)\rfloor$.

We have not formalised the full abstract, relational treatment of the framework by Calcagno et al., but shown the above two properties and the implied frame rule directly by induction on the evaluation of commands.

Fig. 6 gives the page table interface constraints that we promised in Sect. 3. These rules need to be proved about ptable-lift and ptable-trace for a new page table instantiation in order to use the abstract logic presented before. The first two rules are the frame and monotonicity property on ptable-lift. The third rule states that if the domain of ptable-lift does not change, neither does ptable-trace. The last two rules state that updates to the heap outside the trace affect neither the lifting nor the trace of the page table. We have proved the rules in Fig. 6 for the one-level page table instantiation in the examples.

6 Case Study

In this section, we present a small page allocation and assignment routine one might see in operating system services, mapping a *frame*, the physical equivalent of a page, to some address in virtual memory. The program appears in Fig. 7 with simplified syntax. Frame availability information is stored in a frame table,

```
1. fte := ft_free_list;
2. IF fte != NULL THEN
3.      ft_free_list := *ft_free_list;
4.      frame := &fte - &frame_table;
5.      *(ptable + (a2p page_addr)) := f2a frame OR valid_pmask;
6.      ret_val := 0
7. ELSE ret_val := -1
```

Fig. 7. A simple page table manipulating program

which contains one entry per frame in the system, marking it as used or unused. In our program, line 1 attempts to find a free frame's entry in the free frame list. An empty list causes an erroneous return at line 7. Line 3 removes the head of the list. Line 4 calculates the number of the frame from its entry. Upon successfully allocating a frame, line 5 updates the page table with the appropriate entry mapping page-addr to the new frame.

We have proved that the program conforms to the following specification:

$$\{|\mathsf{vars} \wedge^* \mathsf{in\text{-}pt\ page\text{-}addr} \wedge^* \mathsf{frame\text{-}list}\ (f{\cdot}fs) \wedge^* \mathsf{pt\text{-}alloc\ page\text{-}addr} \wedge^* \mathit{ret\text{-}val}_v \mapsto -|\}$$
$$\mathsf{program\ page\text{-}addr}$$
$$\{|\mathsf{vars} \wedge^* \mathsf{in\text{-}pt\ page\text{-}addr} \wedge^* \mathsf{frame\text{-}list}\ fs \wedge^* \mathsf{VPtr}\ f \mapsto - \wedge^* \mathit{ret\text{-}val}_v \mapsto 0 \wedge^* \mathsf{page\text{-}mapped\ page\text{-}addr}|\}$$

where:

$$\begin{aligned}
&\mathsf{vars} \equiv \mathit{ptable}_v \mapsto \mathit{pt} \wedge^* \mathit{frame}_v \mapsto - \wedge^* \mathit{frame\text{-}table}_v \mapsto \mathit{frame\text{-}table} \wedge^* \mathit{fte}_v \mapsto -\\
&\mathsf{in\text{-}pt\ page\text{-}addr}\ (h,\ r) \equiv (\mathsf{VPtr}\ \mathit{pt} + \mathsf{a2p\ page\text{-}addr} \mapsto_v r + \mathsf{a2p\ page\text{-}addr})\ (h,\ r)\\
&\mathsf{pt\text{-}alloc\ page\text{-}addr} \equiv \lambda(h,\ r).\ (r + \mathsf{get\text{-}page\ page\text{-}addr} \rightarrow_p -)\ (h,\ r)\\
&\mathsf{frame\text{-}list}\ \mathit{xs} \equiv \mathsf{list}\ \mathit{xs}\ \mathit{ft\text{-}free\text{-}list}_v\ (\mathsf{fte\text{-}property}\ \mathit{frame\text{-}table})\\
&\mathsf{list}\ []\ h\ P \equiv h \mapsto 0\\
&\mathsf{list}\ (x{\cdot}xs)\ h\ P \equiv \lambda s.\ x \neq 0 \wedge (h \mapsto x \wedge^* \mathsf{list}\ xs\ (\mathsf{VPtr}\ x)\ P \wedge^* P\ x)\ s\\
&\mathsf{fte\text{-}property}\ \mathit{start}\ \mathit{ptr} \equiv \mathsf{entire\text{-}frame}\ (\mathsf{PPtr}\ (\mathsf{f2a}\ (\mathit{ptr} - \mathit{start})))\ \{\rightarrow_p\} -\\
&\mathsf{entire\text{-}frame}\ p \equiv \{p..\mathsf{PPtr}\ (\mathsf{pptr\text{-}val}\ p + 0xFFF)\}
\end{aligned}$$

The functions a2p and f2a convert addresses to page numbers and frame numbers to addresses by respectively dividing and multiplying by 4096.

Since the language in this paper is purely heap based, we use $\mathit{var}_v \mapsto \mathit{var}$ to denote that a variable *var* has a specific value and to represent *var* in Fig. 7. We assume that the page table is accessible from *ptable*, and the frame table lies at *frame-table*. Further, we have a non-empty free list starting at *first-free*, where each address in the list indicates the presence of a frame. Finally, we require that the page table entry that is used to resolve a lookup of page-addr is allocated, and accessible from our address space. We additionally require that page-addr is aligned to the page size. As a result of executing the program, the free frame list becomes shorter and the page at page-addr is fully accessible. The latter is:

$$\begin{aligned}
&\mathsf{page\text{-}mapped}\ \mathit{vp} \equiv \mathsf{entire\text{-}page}\ \mathit{vp}\ \{\mapsto\} - \wedge^* \mathsf{consume\text{-}slices}\ \mathit{vp}\ (\mathcal{U} - \mathsf{entire\text{-}page}\ \mathit{vp})\\
&\mathsf{consume\text{-}slice}\ \mathit{vp}\ \mathit{sl} \equiv \lambda(h,\ r).\ \mathsf{dom}\ h = \mathsf{ptable\text{-}trace}\ (\mathsf{h\text{-}view}\ h\ \mathit{sl})\ r\ \mathit{vp} \times \{\mathit{sl}\}\\
&\mathsf{consume\text{-}slices}\ \mathit{vp}\ S \equiv \mathsf{fold\ op} \wedge^* (\mathsf{consume\text{-}slice}\ \mathit{vp}) \square S\\
&\mathsf{entire\text{-}page}\ \mathit{vp} \equiv \{\mathit{vp}..\mathsf{VPtr}\ (\mathsf{vptr\text{-}val}\ \mathit{vp} + 0xFFF)\}
\end{aligned}$$

All our separation logic statements work on heaps of a precise size. Using up the entire page table entry in our precondition means we must likewise use it all

in the postcondition. A page table entry maps 4096 addresses, but is made up of more slices. Stating that those addresses are mapped does not consume all the slices. The difference in heap size is made up by consume-slices.

The actual mapping-in step from line 5 of our program performs a write to the page table at pt + a2p page-addr. In addition to our assignment rule, we have shown another property that allows us to to conclude from the post-state of line 5 that the page at page-addr is now completely mapped:

$$\begin{array}{l}\text{entire-frame (PPtr } (f2\ frame)) \longrightarrow - \wedge^* \text{ pt-map page-addr } frame \\ \quad \lfloor \longrightarrow \rfloor \text{ page-mapped page-addr}\end{array}$$

$$\begin{array}{l}\text{pt-map page-addr } frame \equiv \\ \lambda(h,\ r).\ (\text{PPtr (pptr-val } r + \text{a2p page-addr}) \mapsto_p \text{f2a } frame \text{ OR } 1)\ (h,\ r)\end{array}$$

This rule is the only place in the case study where we had to unfold page table definitions and reason directly about the encoding. All other reasoning used Separation Logic rules only.

In order to check successful interaction with the newly mapped page, we added an extra segment to our program.

```
*page_addr := 0xFF;
*(page_addr + 3) := *page_addr + 2
```

If executed on a state with offsets 0 and 3 in the page mapped and allocated:

$$\{\!|\text{page-addr} \mapsto - \wedge^* \text{ page-addr} + 3 \mapsto -|\!\}$$

it results in those offsets set to `0xFF` and `0x101`:

$$\{\!|\text{page-addr} \mapsto 0xFF \wedge^* \text{ page-addr} + 3 \mapsto 0x101|\!\}$$

These two offsets are part of the page at page-addr; the program fragment does not change anything else. As we can rewrite

$$\text{page-mapped } p = (p \mapsto - \wedge^* p + 3 \mapsto - \wedge^* \text{ page-mapped } \{p,\ p + 3\}\ p)$$

we can invoke the frame rule and include the rest of the state.

Our case study shows that our logic allows abstract reasoning, even in the presence of page table manipulation. Examples like this would occur when verifying OS code directly. The logic is easier to use for application code that might support sharing, but has no direct access to the page table.

7 Related Work

The primary focus of this work is enhancement of Separation Logic, originally conceived by O'Hearn, Reynolds et al. [9,15]. Separation logic has previously been formalised in mechanised theorem proving systems [1,18]. We enhance these models with the ability to reason about properties on virtual memory.

Previous work in OS kernel verification has involved verifying the virtual memory subsystem [5,8,10]. Reasoning about programs *running* under virtual memory, however, especially the operating systems which control it, remains mostly unexplored. The challenges of reasoning about virtual memory are explored in the development of the Robin micro-hypervisor [16]. Like our work,

the developers of Robin aim to use a single semantics to describe all forms of memory access which simplifies significantly in the well-behaved case. They focus on reasoning about "plain memory" in which no virtual aliasing occurs and split it into read-only and read-write regions, to permit reading the page table while in plain memory. They do not use Separation Logic notation. Our work is more abstract. We do not explicitly define "plain memory". Rather the concept emerges from the requirements and state.

Alkassar et al. [2] have proved the correctness of a kernel page fault handler, albeit not at the Separation Logic level. As in our setting, they use a single level page table and prove that the page fault handler establishes the illusion to the user of a plain memory abstraction, swapping in pages from disk as required. The proof establishes simulation between those two layers, with the full complexity of the page table encoding visible at the lower level for the whole verification. The work presented in this paper focuses on a logic for reasoning about virtual memory instead. The aim is to make such verifications easier and more productive.

Tuch et al. demonstrated the extension of Separation Logic to reasoning about C programs involving pointer manipulation [17]. We believe our framework is orthogonal and can be instantiated readily to the C model.

8 Conclusion and Future Work

We have presented an extension of Separation Logic which allows reasoning about virtual memory and processes running within. The logic fully supports the frame rule as well as the other Separation Logic proof rules and allows for a convenient representation of predicates on memory at three levels: the virtual map, physical heap and virtual address space. We have presented a small case study that demonstrates the applicability of the logic to OS level page table code as well as client code using the page table mechanism.

For analysing the logic in this paper we chose a simplified machine and page table instance. The logic does not depend on the implementation of either.

We have significantly extended our previous work on virtual memory and have managed to fully hide the complexity of virtual memory reasoning for code that does not directly modify the page table. We also have shown that high-level reasoning is still possible for code that does. The concepts in the logic are close to the mental model kernel programmers have of virtual memory.

Practical implementations of virtual memory in hardware use a cache, the translation lookaside buffer (TLB). To include this concept in our model, we could add standard cache consistency protocols such as TLB flushes when appropriate.

The next steps are to fully instantiate this model to the C programming language and apply it to the verification of the seL4 microkernel.

Acknowledgements. We thank Michael Norrish, Harvey Tuch, Simon Winwood, and Thomas Sewell for discussions and comments on earlier versions of this paper as well as Hongsoek Yang for sharing his insight into Separation Logic.

References

1. Affeldt, R., Marti, N.: Separation logic in Coq (2008), http://savannah.nongnu.org/projects/seplog
2. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963. Springer, Heidelberg (to appear, 2008)
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL 2005: Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 259–270. ACM, New York (2005)
4. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS 2007: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science, pp. 366–378. IEEE Computer Society Press, Los Alamitos (2007)
5. Dalinger, I., Hillebrand, M.A., Paul, W.J.: On the verification of memory management mechanisms. In: Borrione, D., Paul, W.J. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 301–316. Springer, Heidelberg (2005)
6. Derrin, P., Elphinstone, K., Klein, G., Cock, D., Chakravarty, M.M.T.: Running the manual: An approach to high-assurance microkernel development. In: Proc. ACM SIGPLAN Haskell WS, Portland, OR, USA (September 2006)
7. Elphinstone, K., Klein, G., Derrin, P., Roscoe, T., Heiser, G.: Towards a practical, verified kernel. In: Proc. 11th Workshop on Hot Topics in Operating Systems, San Diego, CA, USA, p. 6 (May 2007)
8. Hillebrand, M.: Address Spaces and Virtual Memory: Specification, Implementation, and Correctness. PhD thesis, Saarland University, Saarbrücken (2005)
9. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL 2001: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 14–26. ACM, New York (2001)
10. Klein, G., Tuch, H.: Towards verified virtual memory in L4. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223. Springer, Heidelberg (2004)
11. Kolanski, R.: A logic for virtual memory. In: Huuck, R., Klein, G., Schlich, B. (eds.) Proc. 3rd Int'l Workshop on Systems Software Verification (SSV 2008). ENTCS, pp. 55–70. Elsevier, Amsterdam (to appear, 2008)
12. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
13. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL 2004: Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 268–280. ACM, New York (2004)
14. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL 2005: Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 247–258. ACM, New York (2005)
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. 17th IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002)
16. Tews, H.: Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. In: C/C++ Verification Workshop, Technical Report ICIS-R07015, pp. 59–68, Oxford, UK. Radboud University Nijmegen (July 2007)
17. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hofmann, M., Felleisen, M. (eds.) POPL 2007, pp. 97–108. ACM, New York (2007)
18. Weber, T.: Towards mechanized program verification with separation logic. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210. pp. 250–264. Springer, Heidelberg (2004)

Unguessable Atoms: A Logical Foundation for Security

Mark Bickford

ATC-NY and Cornell University Computer Science
33 Thornwood Dr. Suite 500, Ithaca, NY, USA
markb@cs.cornell.edu

Abstract. We show how a type of atoms, which behave like urelements, and a new proposition that expresses the independence of a term from an atom can be added to any logical system after imposing minor restrictions on definitions and computations. Working in constructive type theory, we give rules for the independence proposition and show how cryptographic protocols can be modeled as automata exchanging atoms. This model provides a unifying framework for reasoning about security and allows us to combine a general model of computation with a simple model of acquisition of secret information. As an application, we prove a fundamental property of nonces that justifies the axioms for nonces used in the protocol composition logic (PCL) of Datta, Derek, Mitchell and Roy. The example shows that basic security properties are naturally expressed in terms of independence and the causal ordering of events. The rules and example proofs are fully implemented in the Nuprl proof development system.

1 Introduction

Formal models of security depend on three key ingredients: *agents* in a *computation system*, a representation of *protected information*, and a definition of information *learning*. In a *symbolic security model* like the Dolev-Yao model [9] or the models based on multi-set rewriting systems [10], system states are represented as terms or sets of terms, agents can perform only a limited set of rewrite rules on these states, and an adversary is deemed to learn a piece of protected information merely from his ability to generate its representation. The limited computation system of a symbolic model makes it impossible for an adversary to simply *guess* the protected information.

In contrast, in *analytic security model* the computation system is a general model of distributed computation (interacting Turing machines, process calculus, etc.), agents can be any programs or processes (with some resource bounds e.g. polynomial time), and protected information can be any function of the state of an agent. An adversary can generate arbitrary values but is not deemed to have *learned* a piece of protected information unless it can produce a value that is significantly more likely to be correct than a random guess. The formal definition of learning in an analytic model is typically in terms of probabilistic games (see [12], for example).

We would like a way to specify and prove security properties that has the generality of the analytic model and some of the simplicity of the symbolic model. We would like, as in the analytic model, to define agents as "all distributed programs" and to carry out our proofs in a *general purpose logic of distributed systems*. By doing this, our security

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 30–53, 2008.

theorems have a greater significance since we will be proving impossibility results for adversaries in the general computation system rather than the limited symbolic computation system. We will also be using the same logical framework for all proofs about programs–security proofs will not be done in a special logical system.

To avoid the necessity for a formal theory of polynomial time complexity, probability, and games, or simply to have a more abstract theory, we also want the simplicity of the symbolic model, in that protected information can be represented in a way that allows us to model *learning* the information simply as *acquiring* the information. Thus, in particular, the protected information must be *unguessable*; for if an adversary had a procedure that could generate guesses that eventually included all protected information, then we would have to resort to modeling the fact that the adversary does not *know* which guesses are correct.

How can this be possible? The secure agents must represent protected information in their state using some data-type. A theory of "all programs" must allow even resource bounded programs to apply any computable function. Wouldn't any data-type have some finite representation and therefore would its set of values not be enumerable by a computable function?

1.1 Unguessable Atoms

Our solution to this dilemma is to represent protected information using a new type $Atom$. The members of type $Atom$, called *atoms*, are terms that evaluate to canonical forms $tok(a)$, $tok(b)$, ..., called tokens, parameterized by names a,b,.... But, other than our ability to tell one from another, the name parameters have no significance in the logic. This is made explicit by the *permutation rule* that says

> From any judgement, $J(a, b, \dots)$, in the logic, mentioning a finite set $a, b, \dots$ of these names, we may infer the truth of $J(a', b', \dots)$ whenever the mapping $a \mapsto a', b \mapsto b', \dots$ is a permutation.

We add to the computation system a primitive, 4-place operation $acase$ and a computation rule that says that $acase(tok(a), tok(b), x, y)$ computes to y when a and b are distinct names and computes to x when a and b are the same name. Since every atom evaluates to a canonical form, the $acase$ computation decides whether two atoms are equal.

To make the permutation rule valid, we must constrain the logic in two ways. The computation rule for $acase$ preserves the property that if σ is a permutation of names and t computes to t' then $\sigma(t)$ computes to $\sigma(t')$. The first constraint on the logic is that every additional computation rule must preserve this property. This constraint means that the operator $acase$ is, essentially, the only non-trivial computation on atoms that can be added to the logical system.

Second, the definition mechanism must be constrained so that the names $a, b, \dots$ are *unhideable*. This means that a definition like $f(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$ is disallowed because the names a and b occur on the righthand side of the definition but not on the lefthand side. If this were allowed, the permutation rule would be inconsistent: we could prove a judgement that $f(1) = tok(a)$, and use the permutation rule to conclude $f(1) = tok(b)$ and then conclude that $tok(a) = tok(b)$, from which,

using the $acase$ computation, we may deduce $0 = 1$. Thus, every name mentioned in the righthand side of a definition must also be included among the parameters on the lefthand side, so $f\{a,b\}(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$ is an allowed definition. (Permutation of $f\{a,b\}(1) = tok(a)$ to $f\{b,a\}(1) = tok(b)$ now leads to no contradiction.)

These changes to the logic are not "intrusive", at least for the NuPrl system. Since NuPrl terms already had parameters (such as integer values) we only had to add an additional parameter type. The restriction on definitions easy to implement and it makes no restriction on the preexisting system since the preexisting terms do not mention the new token parameters. NuPrl was designed to be an open system, so adding new rules is always easy.

Note that for any finite set of atoms A there is still a computable function f that enumerates A, but any such function f is essentially a table lookup like $f\{a,b\}(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$. An adversary using the function f to "guess" atoms must have the function f available in its state or in its program, and since the names a and b are unhideable, they are "in" f already so the agent already "has" them. Thus, a table lookup function like f is of no use in guessing atoms since the only atoms it can guess are atoms that it already has. This is the sense in which we say that atoms are *unguessable*. Also, note that "bare" names are not terms, so there is no type of names, and no mapping $a \rightarrow tok(a)$ from names to atoms.

The semantics of atoms is presented in [2]. That paper explains how the *unhideable tokens* are added to the terms and how a *supervaluation* semantics for atoms justifies the permutation rule. This paper depends only on the properties of atoms we have mentioned and the fact that the set of names *mentioned by* a closed term t is unambiguous.

Thesis. Our thesis is that by making the constraints on the logical system necessary to introduce the $Atom$ type we will be able to have the best of both kinds of security models. We will be able to introduce a type of data values (atoms) that are "unguessable," relieving us from the need for a distinction between learning an atom and acquiring an atom. The theory of "all programs" is just as it was before the introduction of atoms, and programs can now compute with atoms as well, but cannot use any hidden structure of atoms, not because of a restriction on the concept of programs, but because of a global constraint on the mathematical framework. Thus, the simplicity of the concepts of protected information and learning of protected information in the symbolic model can be combined with the generality of the computation system in the analytic model.

Adding atoms to our computations is only the first step in building a model in which the representation of protected information is *atoms*, whose computation system is a general model of distributed programs called *message automata*, and defines learning (= acquiring) in terms of a proposition called *independence*, which is the topic of the next section.

The work on Nominal Logic by Gabbay and Pitts [11] introduces names as *urelements* of Fraenkel-Mostowski set theory. Urelements are also used in the work on Choiceless Polynomial Time by Blass, Gurevich and Shelah [5]. Clearly, there is a close relation between terms with atoms and hereditarily finite sets with urelements, so the novelty of our model and methods is not a consequence of the use of atoms alone. We will discuss a suite of concepts that includes *atoms*, *independence*, *event structures*

and a compositional semantics for distributed programs with *frame conditions*. Using these ingredients we can model fundamental security mechanisms and elegantly state and prove general theorems that characterize them.

2 Independence

Intuitively, an agent will have learned an atom a at some point in its history if (a component of) one of its state variables has value a. The semantics of our computation system includes an expression (**state after** e) that denotes the state of an agent after an event e has occurred. Events e are essentially points in space/time, so if we can define what it means for an expression like (**state after** e) to contain an atom a, then we can use that to define when an agent has acquired atom a. We have discovered that in a logic based on *constructive type theory* the rules for the primitive proposition $(x{:}T\|a)$, saying that an expression x of type T *does not* contain atom a, are simpler than the rules for its negation (which says that x *does* contain a). We read $(x{:}T\|a)$ as "x of type T is *independent of* atom a". When the type T is clear from context we may write $x\|a$ and say that "x and a are independent".

The rules for independence currently implemented in Nuprl are listed in table 1.

Table 1. Rules for Independence

INDEPENDENTTRIVIALITY
closed x mentions no names $\quad H \vdash x \in T \quad H \vdash a \in Atom$
$$\overline{H \vdash (x{:}T\|a)}$$

INDEPENDENTBASE
$$\frac{H \vdash \neg(x = a \in Atom)}{H \vdash (x{:}Atom\|a)}$$

INDEPENDENTABSURDITY
$$\overline{H, (a{:}Atom\|a), J \vdash C}$$

INDEPENDENTAPPLICATION
$$\frac{H \vdash (f{:}(v : A \to B)\|a) \quad H \vdash (x{:}A\|a)}{H \vdash (f(x){:}B[x/v]\|a)}$$

INDEPENDENTEQUALITY
$$\frac{H \vdash T_1 = T_2 \in Type \quad H \vdash x_1 = x_2 \in T_1 \quad H \vdash a_1 = a_2 \in Atom}{H \vdash (x_1{:}T_1\|a_1) = (x_2{:}T_2\|a_2) \in Type}$$

INDEPENDENTSUPERTYPE
$$\frac{H \vdash (x{:}T'\|a) \quad H \vdash T' \subseteq T}{H \vdash (x{:}T\|a)}$$

INDEPENDENTSET
$$\frac{H \vdash (x{:}T\|a) \quad H \vdash x \in \{v : T|\ P\}}{H \vdash (x{:}\{v : T|\ P\}\|a)}$$

The triviality rule, "independentTriviality" says that a closed term that mentions no names at all is independent, as a member of any type that contains it, of any atom. The base rule "independentBase" says that distinct atoms are independent, and the "independentAbsurdity" rule says that an atom is not independent of itself.

The application rule "independentApplication" is crucial; it says that if function f and argument x are both independent of atom a then the application $f(x)$ is also independent of atom a. This rule formally captures the fact that atoms can not be built from pieces (or from nothing).

The equality rule "independentEquality" says that independence is well-defined from its constituent pieces, so that these pieces can be replaced by equivalent pieces without changing the meaning. Rules of this form are needed because in constructive type theory terms may have many different types and two terms may stand for the same member of one type but different members of another type. Finally, the rules "independentSet" and "independentSupertype" are somewhat technical and concern the relation between independence and subtypes. The supertype rule says that if x is independent of a as a member of type T' then it is independent of a as a member of any supertype T. The corresponding subtype rule is not valid in general, but for subtypes formed using Nuprl's *Set type* the rule is valid. Thus, the set rule says that if x is independent of a as a member of type T, then it is also independent of a as a member of any subset of T that contains x.

As a simple example of what follows from these rules, let us prove that integers and atoms are independent.

Lemma 1

$$\forall a : Atom.\, \forall i : \mathbb{Z}.\, (i{:}\mathbb{Z} \| a)$$

Proof. By induction on i: the case $i = 0$ is $(0{:}\mathbb{Z} \| a)$ which follows by using "independentTriviality" since 0 is closed and mentions no names.

Now assume $(i{:}\mathbb{Z} \| a)$ and show that both $i + 1$ and $i - 1$ are independent of atom a:

Since $i + 1$ is $(\lambda x.\ x + 1)(i)$, using the "independentApplication" rule it is enough to show $(\lambda x.\ x + 1{:}\mathbb{Z} \to \mathbb{Z} \| a)$ and $(i{:}\mathbb{Z} \| a)$. The former again follows trivially since $\lambda x.\ x + 1$ is closed and mentions no names, and the latter is the induction hypothesis. The $i - 1$ case is the same. □

Since product types, union types, and list types are built using constructors $\lambda x \lambda y.\langle x, y \rangle$, $\lambda x.\mathrm{inl}(x)$, $\lambda x.\mathrm{inr}(x)$, **nil** and $\lambda x \lambda y.x$ **cons** y, all of which are trivially independent of atoms, lemmas similar to lemma 1 can be proved for many types. The rules in Table 1 suffice for proving independence assertions $(x{:}T \| a)$ where T is any algebraic data type, but the rules may not be complete for other types (such as general function types).

Types T_1 and T_2 are *extensionally equal*, written $T_1 \equiv T_2$, if $T_1 \subseteq T_2$ and $T_2 \subseteq T_1$. The following is an immediate corollary of the supertype rule.

Corollary 1. $T_1 \equiv T_2 \ \Rightarrow\ (x{:}T_1 \| a) \Leftrightarrow (x{:}T_2 \| a)$

2.1 Definition of Independence

In appendix A we give the full meta-theoretic definition of the new independence proposition in Martin-Löf's semantics for constructive type theory and we show how the inference rules are proved in the meta-theory. The truth of an independence proposition is defined as follows:

Definition 1. *The proposition* $(x{:}T\|a)$ *is true if and only if* a *evaluates to* $tok(b)$ *for some name* b*, and there exists a term* y *in type* T *such that* $x = y \in T$ *and* y *does not mention name* b*.*

This definition deserves some comment. If we were working with untyped terms only, then terms containing name parameters would be essentially isomorphic to hereditarily finite sets with urelements. Then independence of x and $tok(b)$ would mean only that b is not in the transitive closure of x, or, in our language, that term x does not mention b. This is not a type-theoretic proposition since the term $(\lambda x.\lambda y.y)(tok(b), 3)$ mentions b but computes to 3 which does not mention b. Propositions in type theory are types, and types must be closed under computation.

Our typed independence proposition has even stronger closure properties. The type $(x{:}T\|a)$ is invariant under substitutions $x = x' \in T$. Consider, for example, the type $T = \{0, \ldots, n\} \to \mathbb{Z}$. For any term $f \in T$, whether it mentions atoms or not, we have $(f{:}T\|a)$, because equality in T is extensional and f is extensionally equal to a function g expressed in terms of a finite table of numbers.

2.2 Inherence

To highlight some of the subtle constraints on a type-theoretic definition of $(x{:}T\|a)$, consider its negation which we write, $(x{:}T \gg a)$, which means that atom a is *inherent* in x. We tried, and failed to base the theory on this primitive. Doing so would seem to require a rule dual to 'independentApplication", the following *inherence application principle*

$$(f(x){:}B[x/v] \gg a) \Rightarrow$$
$$((f{:}(v : A \to B[v]) \gg a) \ \vee \ (x{:}A \gg a))$$

i.e. if an atom is inherent in the result of applying a function f to an argument x then the atom is either inherent in the function f or else inherent in the argument x.

But that rule is not constructively valid, as will be shown in Appendix B. The reason is that a witness to the inherence application principle, would provide an effective procedure to determine whether an atom inherent in $f(x)$ is inherent in f or in x. We considered various additions to the computation system that would allow the tracing of atoms during evaluation and which would allow us to construct such a witness, but no mechanism we considered was compatible with the permutation semantics on atoms and the other constraints on the logical system. We argue in Appendix B that no such mechanism is possible.

Once we realized that independence was a sufficient basis for security properties in our logic we abandoned the notion of inherence and restated all security specifications in terms of independence. In a classical logic, the inherence application principle is equivalent to the independentApplication rule, so users of classical logic could use either inherence or independence as a primitive. The ability to add a type of atoms satisfying the permutation semantics does not depend on the logic being constructive, so our whole approach is compatible with classical logic. Atoms and independence could be easily added to logical frameworks such as Twelf [19] or Isabelle/HOL [16].

3 Event Structures

The semantics of our computation system is expressed in the language of *event structures*. We have several reports [3, 4] on the semantics of *message automata* and how distributed programs can be extracted from event logic proofs. In this paper we will focus on recent additions to this framework that make it possible to state and prove security properties.

An *event language* $\mathcal{L}$ is any language extending $\langle E, loc, < \rangle$, where loc is a function on E, and $<$ is a relation on E. For an event $e \in E$, $loc(e)$ is the *location* of the event and $e_1 < e_2$ means event e_1 causally precedes event e_2. Locations in the event structure represent agents, processes, or threads. An *event structure* is a model for $\mathcal{L}$ that satisfies the axioms

- $\leq$ is a locally-finite partial order (every e has finitely many predecessors)
- *Local order*, defined by $e_1 <_{loc} e_2 \equiv e_1 < e_2 \ \wedge \ loc(e_1) = loc(e_2)$, is a total ordering on any set of events whose members all have the same location.

An event structure represents a possible history of the universe—including, when modeling a security protocol, both the honest participants and the "environment" of adversaries. We will define a program as a set of constraints, and the semantics of a program as the collection of event structures consistent with those constraints. (From a logician's point of view a program is a finitely axiomatizable theory.)

In order to provide a semantics for message automata we extend the core event language with operators $kind$, val, $sender$, **when**, and **after**. The axioms enforce a model in which messages are sent reliably on named, fifo, links. Each message has an associated tag, which can be thought of as header information. Every event has an associated kind and value. The receipt of a message with tag tg on link l is an event e of kind $rcv(l, tg)$ and its value is the message received (whose type depends on the link and tag) and $sender(e) < e$ is the event that sent the message. All other events are called internal events. Assigning kinds and values to internal events is a technical convenience. In particular, it provides a convenient way to represent nondeterministic choice: the value of an internal event can be chosen nondeterministically. It is important that our model of an adversary includes nondeterminism.

When event e occurs, at location $i = loc(e)$, the value of state variable x is $(x \ \textbf{when}\ e)$ and the whole state of agent i is

$$\textbf{state when}\ e \equiv (\lambda x.\ x\ \textbf{when}\ e)$$

Event e may affect the state, and after e has occurred the state is

$$\textbf{state after}\ e \equiv (\lambda x.\ x\ \textbf{after}\ e)$$

3.1 Extended Event Structures

We will write some programs explicitly (e.g., the programs representing honest participants) and we provide the notation of message automata to do that conveniently. Other programs represent unknown adversaries, and we need some way to characterize our

assumptions about those. In particular, we want to say that an adversary did not "have" an atom a initially. The atom a could be part of the initial state of the unknown agent or it part of the program that it executes to update its state, send messages, or assign values to internal events.

We represent the initial state and program of arbitrary agents in the event structure by adding new operators **Init**, **Trans**, **Send**, and **IVal**, and axioms that relate them to the other operators as follows:

- If e is the first event at its location,
  state when $e = \mathbf{Init}(loc(e))$
- For an event e of kind k and value v,
  state after $e = \mathbf{Trans}(k, v, \mathbf{state\ when}\ e)$
- For any receive event e' on link l with tag tg, and $sender(e') = e$,
  $\langle l, tg, val(e')\rangle \in \mathbf{Send}(kind(e), val(e), \mathbf{state\ when}\ e)$
- For any internal event e of kind k,
  $\exists n{:}\mathbb{N}.\ val(e) = \mathbf{IVal}(k, n, \mathbf{state\ when}\ e)$.

Thus **Init** and **Trans** abstractly define how the state of every agent evolves, and this part of every agent is deterministic. The operator **Send** gives, again deterministically, the list of messages that an agent sends in response to an event. Finally, IVal deterministically computes the value of an internal event based on two things: the current state and a nondeterministically chosen natural number (hence the existential quantification in the axiom). We want to allow nondeterminism, but complete nondeterminism would allow an adversary to guess an atom.

3.2 Atom Dataflow Lemmas

All of the theorems we discuss here have been proved in Nuprl, and all the reasoning about independence reduces to the rules in table 1 using automated tactics. In particular, the type parameters T in propositions $(x{:}T\|a)$ are usually supplied automatically by a type inference algorithm. Thus, for readability, we will suppress the type parameter in these expressions for the remainder.

Trans, **Init**, **Send**, and **IVal** abstractly define the *program* at each agent i. By $\mathbf{IVal}(k)$ we mean $\lambda n, s.\ \mathbf{IVal}(k, n, s)$, and similarly for other Curried application terms. A kind k has location i if it is either an internal action at i or a receive on a link with destination i.

Definition 2. *If* $\boldsymbol{IVal}(k)$, $\boldsymbol{Trans}(k)$ *and* $\boldsymbol{Send}(k)$ *are independent of* a *for every kind* k *with location* i*, then we say that the program at location* i *is independent of* a *and write* $\mathbf{Program}(i)\|a$.

If $\mathbf{Program}(i)\|a$ and $\mathbf{Init}(i)\|a$ then the agent at location i does not initially "have" atom a. The fundamental lemma asserts

> If an agent did not have an atom initially and has not received the atom then it does not have the atom.

Stated formally, we have:

Lemma 2 (Fundamental Atom Lemma). *For all events* e *and atoms* a*, if* **Program**$(loc(e))\|a$ *and* **Init**$(loc(e))\|a$ *and if*

$$\forall e' <_{loc} e.\ isrcv(e') \ \Rightarrow\ (val(e')\|a)$$

then (**state when** $e\|a$)

Proof. By induction on the well-founded relation $<_{loc}$. When e is the first event, **state when** $e =$ **Init**$(loc(e))$, which is independent of a by assumption. Otherwise, let p be the local predecessor of e, then

$$\textbf{state when}\ e = \textbf{Trans}(kind(p), val(p), \textbf{state when}\ p)$$

By the induction hypothesis, (**state when** $p)\|a$, and **Trans**$(kind(p))\|a$ by assumption. So we use the independentApplication rule to reduce this case to proving that $val(p)\|a$. If p is a receive event, then this follows from the assumptions, but if p is an internal event, then

$$val(p) = \textbf{IVal}(kind(e), n, \textbf{state when}\ p)$$

for some number n. We again use the independentApplication rule to show that since all of the pieces are independent of a either by induction, assumption, or by lemma 1, **IVal**$(kind(e), n,$ **state when** $p)$ is also independent of a. □

Definition 3. *We use the notation* $sends(e)\|a$ *as a shorthand for*

$$\forall e' : E.\ (isrcv(e') \ \wedge\ sender(e') = e) \ \Rightarrow\ val(e')\|a$$

which says that all messages sent by event e *are independent of atom* a*.*

Corollary 2. *For all events* e *and atoms* a*, if* **Program**$(loc(e))\|a$ *and* **Init**$(loc(e))\|a$ *and if*

$$\forall e' \leq_{loc} e.\ isrcv(e') \ \Rightarrow\ (val(e')\|a)$$

then $val(e)\|a$ *and* $sends(e)\|a$

Proof. Under the given assumptions, $(val(e)\|a)$ is proved as in the proof of Lemma 2. If e' is a receive on link l with tag tg, and with $sender(e') = e$, then,

$$\langle l, tg, val(e')\rangle \in \textbf{Send}(kind(e), val(e), \textbf{state when}\ e)$$

We already have $(val(e)\|a)$, and, by Lemma 2 , **state when** $e\|a$. Thus, since **Program**$(loc(e))\|a$, we may use the independentApplication rule to deduce that the list **Send**$(kind(e), val(e),$ **state when** $e)$ is independent of a, and hence that $val(e')$ (a component of one of its elements) is also independent of atom a. □

4 Modeling Cryptographic Systems

Atoms, independence, and message automata provide the logical foundation for our formal account of security, but to reason about real cryptographic protocols we must build on this foundation another "layer" to model the ability of agents to choose nonces, encrypt messages, establish a public/private key pairs, and perform other more complex cryptographic operations such as secret shares.

We start by discussing how we model the simplest of these, the ability of agents to choose nonces. This example will serve several purposes. It will illustrate a general approach in which we model a cryptographic service as a *virtual server* represented by a message automaton. Lacking the space for a full formal definition of message automata, we use this opportunity to explain them by example. The example also illustrates the elegance of the formal specification of nonces in terms of causal order and independence.

4.1 Nonces

To model the ability of agents to generate nonces, we provide each agent i with a virtual server s_i with which it communicates on link l_i from s_i to i and l^i from i to s_i. The server will have a list *ats* of atoms with no repeats, and a pointer into the list. When agent i wants a nonce, it sends a message with tag $nonce$ on link l^i. We call receipt of this message by server s_i a *nonce event* and let $E(\mathit{Nonce}_i)$ be the set of such events. In response to an event $n \in E(\mathit{Nonce}_i)$ server s_i sends an atom, $\mathit{Nonce}_i(n)$, on link l_i also with tag $nonce$ by sending the next atom on the list and advancing the pointer. We will implement this behavior with a message automaton and show that, assuming the atoms in *ats* are independent from all agents, the server realizes the following abstract specification for nonces.

Definition 4. *A partial function Nonce_i from events to atoms with domain $E(\mathit{Nonce}_i)$ is a* nonce interface *if for all $n \in E(\mathit{Nonce}_i)$ and any event e*

$$n \not\leq e \Rightarrow (val(e) \| \mathit{Nonce}_i(n))$$

This says that unless an event is causally after the generation of a nonce, its value cannot contain the nonce.

4.2 Nonce Server

The nonce server is a message automaton consisting of the ten clauses listed in table 2. In general, a message automaton is simply a finite set of such clauses. The semantics of each clause is a sentence in the language of (extended) event structures, and the semantics of a message automaton is the set of event structure that satisfy all of its constraints. We call the first four clauses in the example *active clauses* and the remaining six clauses *frame clauses*. The active clauses cause the program to initialize and update state variables and send messages. The frame clauses limit what other active clauses may be added later and thus define the class of automata that may be composed with this one.

Table 2. Clauses of the Nonce Server

NonceServer(i,*ats*) $\equiv$ let k = rcv(l^i, $nonce$) in

at location s_i **initially** ptr = 0
at location s_i **initially** L = ats
effect of k **is** ptr:=ptr+1
k **sends on link** l_i :
if $ptr < len(L)$ then $\langle nonce, L[ptr]\rangle$ else **nil**

only [ ] **affects** L **at location** s_i
only [k] **affects** ptr **at location** s_i
k **affects only** [ptr] **at location** s_i
k **sends only on link/tags** [$\langle l_i, nonce\rangle$]
only k **sends on link/tags** [$\langle l_i, nonce\rangle$]
only [k] **read** L **at location** s_i

The semantics of the first four (active) clauses in the nonce server is the conjunction of the following four constraints (on any events e and e'):

$$(loc(e) = s_i \ \wedge \ first(e)) \ \Rightarrow \ ptr \ \textbf{when}\ e = 0 \tag{1}$$

$$(loc(e) = s_i \ \wedge \ first(e)) \ \Rightarrow \ L \ \textbf{when}\ e = ats \tag{2}$$

$$kind(e) = k \ \Rightarrow \ ptr \ \textbf{after}\ e = (ptr \ \textbf{when}\ e) + 1 \tag{3}$$

$$(\mathit{kind}(n) = rcv(l_i, nonce) \Rightarrow$$
$$(e = \mathit{sender}(n) \ \wedge \ kind(e) = k) \Rightarrow \tag{4}$$
$$\begin{pmatrix} ptr \ \textbf{when}\ e < len(L \ \textbf{when}\ e) \wedge \\ val(n) = (L \ \textbf{when}\ e)[ptr \ \textbf{when}\ e] \wedge \\ \forall n'.\ \mathit{kind}(n') = rcv(l_i, nonce) \ \wedge \ \mathit{sender}(n') = e \\ \Rightarrow n' = n \end{pmatrix}$$

The semantics of the first five of the six frame clauses in the nonce server is conjunction of the following constraints. For any event e and any state variable x

$$L \ \textbf{after}\ e = L \ \textbf{when}\ e \tag{5}$$

$$kind(e) \neq k \ \Rightarrow \ ptr \ \textbf{after}\ e = ptr \ \textbf{when}\ e \tag{6}$$

$$kind(e) = k \ \wedge \ x \neq ptr \ \Rightarrow \ x \ \textbf{after}\ e = x \ \textbf{when}\ e \tag{7}$$

$$(isrcv(e) \ \wedge \ kind(sender(e)) = k) \ \Rightarrow$$
$$(link(e) = l_i \ \wedge \ tag(e) = nonce) \tag{8}$$

$$kind(e) = rcv(l_i, nonce) \ \Rightarrow \ kind(\mathit{sender}(e)) = k \tag{9}$$

The clause k **sends only on link/tags** [$\langle l_i, nonce\rangle$] and its semantics, constraint 8, deserve comment. The clause says that messages sent by events of kind k can only be received on link l_i with the given tag. In general, in the absence of such a constraint, if an agent sends a message on a link l, it may also send the same message on other links, and this is our model of eavesdropping by adversaries. We simply note that for

messages sent by honest agents on unprotected network links we have no constraint that keeps the agent from broadcasting the message on links to the adversary. In this logical approach, the possibility of eavesdropping is modeled as a lack of a prohibition against it. Thus, the presence of the clause under discussion in the nonce server implies that the link l_i between server s_i and location i is a *secure link* that does not permit eavesdropping.

The final frame clause is **only** $[k]$ **read** L **at location** s_i. We call this a *read-frame clause*. Read frame clauses are a new feature of message automata and are needed to constrain honest agents to limit access to protected information. This also deserves comment. When we write the clauses of an honest agent, we do not assume that those clauses will be all of the code at that location. We assume only that other code that runs at that location will obey the frame conditions. This means that our model of adversaries includes code running at the same location as the honest agent, as long as it satisfies the frame conditions. This means that a protocol that depends on keeping some information confidential must include read-frame constraints.

The semantics of a read-frame clause is that the actions done in response to an event of kind k that may not read state variable x must be oblivious to the value of x. This means that for two states s_1 and s_2 that differ only on x, which we write as $s_1 = s_2$ mod x, the program described by **Trans**, **Send**, and **IVal** does the same things in response to kind k in state s_2 as it does in state s_1. To formalize this we define $Same_x(k, s_1, s_2)$ to be

$$\begin{aligned}&\forall v.\ \mathbf{Trans}(k, v, s_1) = \mathbf{Trans}(k, v, s_2)\ \mathbf{mod}\ x\\ &\wedge\ \mathbf{Send}(k, v, s_1) = \mathbf{Send}(k, v, s_2)\\ &\wedge\ \forall n.\ \mathbf{IVal}(k, n, s_1) = \mathbf{IVal}(k, n, s_2)\end{aligned}$$

Then the semantics of **only** $[k]$ **read** L **at location** s_i is

$$(k' \neq k\ \wedge\ s_1 = s_2\ \mathbf{mod}\ L)\ \Rightarrow\ Same_L(k', s_1, s_2) \tag{10}$$

If e is an event at location s_i, the location of the nonce server, then **state when** e, the state of the nonce server when event e occurs, includes $(L\ \mathbf{when}\ e)$, the list of nonces. We need an expression for the state of the nonce server except for the list of nonces. This can be expressed by $(\mathbf{state\ when}\ e) \setminus L$ which is the state with L set to the nil list, i.e. the state defined by

$$\lambda x.\ \mathbf{if}\ x = L\ \mathbf{then\ nil\ else}\ (x\ \mathbf{when}\ e)$$

The nonce assumption is that the initial list *ats* of nonces has no repeats, and the program of any agent (including the nonce server itself) is independent of the atoms in *ats*. Also, the initial state of every agent is independent of the atoms in *ats*, except for the nonce server itself, where we assume that the initial state except for L is independent of the atoms in *ats*. Thus the nonce assumption is the conjunction of the following four constraints, for any $a \in ats$:

$$\forall n, m < len(ats).\ ats[n] = ats[m]\ \Rightarrow\ n = m \tag{11}$$

$$\forall j.\ \mathbf{Program}(j) \| a \tag{12}$$

$$\forall j \neq s_i.\ \mathbf{Init}(j) \| a \tag{13}$$
$$\mathbf{Init}(s_i) \setminus L \| a \tag{14}$$

We define the nonce events $E(Nonce_i)$ to be the events that are receipts from the nonce server. Thus, the set $E(Nonce_i)$ is the subtype defined by

$$\{e : E |\ kind(e) = rcv(l_i, nonce)\}$$

For $e \in E(Nonce_i)$ the value $Nonce_i(e)$ is defined to be $val(e)$, the message received. By constraints 9 and 4 this value is

$$(L\ \mathbf{when}\ sender(e))[ptr\ \mathbf{when}\ sender(e)]$$

We will show that the constraints 1–10 and assumptions 11–14 imply that $Nonce_i$ is a nonce interface:

$$\forall n : E(Nonce_i)\ \forall e : E\ n \not\leq e \Rightarrow val(e) \| Nonce_i(n)$$

This will follow as a corollary of the following stronger property that we can prove by induction.

Theorem 1 (Fundamental Nonce Property). *For every nonce event $n \in E(Nonce)$ and every event e, $n \not\leq e$, not causally after n, the atom $a = Nonce(n)$ satisfies the following independence assertions:*

$$(a)\ \ loc(e) \neq s_i\ \Rightarrow\ (\mathbf{state}\ \mathbf{when}\ e \| a)$$
$$(b)\ \ loc(e) = s_i\ \Rightarrow\ ((\mathbf{state}\ \mathbf{when}\ e) \setminus L \| a)$$
$$(c)\ \ val(e) \| a$$
$$(d)\ \ sends(e) \| a\ \vee e = sender(n)$$

Proof. The proof is by causal induction on e. Assuming that $n \not\leq e$, we may assume that (a)–(d) hold for all $e' < e$ (since, in that case, $n \not\leq e'$). Note first that from constraints (2) and (5) it follows easily, by induction on $<_{loc}$, that for any event ev at location i, $(L\ \mathbf{when}\ ev) = ats$, i.e. the state variable L is held constant. Since $n \in E(Nonce_i)$ we have

$$a = Nonce_i(n) = ats[ptr\ \mathbf{when}\ sender(n)]$$

Thus $a \in ats$.

By part (d) of the induction hypothesis, all messages received before or at e are independent of a. This is because, for a receive event $e' \leq e$, $sender(e') < e$ so by (d) either $sender(e') = sender(n)$ or else $val(e') \| a$. But, by constraints 9 and 4, if $sender(e') = sender(n)$ then $e' = n$ and hence $n < e$, contrary to our assumption.

We consider first the case $loc(e) \neq s_i$. In this case, part (b) is trivial, and part (a) follows from Lemma 2 because the initial state is independent of $a \in ats$ by assumption (13) and all messages received before or at event e are independent of a. Parts (c) and (d) follow similarly from Corollary 2.

So, we may assume that $loc(e) = s_i$, i.e. e is an event at the nonce server. Part (a) is now trivial. Parts (b), (c), and (d) no longer follow from Lemma 2 and its corollary

because the initial state at location s_i is not independent of atom a. Any event ev with $loc(ev) = s_i$, other than a nonce request, i.e. for which $kind(ev) \neq k$, does not read state variable L. So, constraint (10) implies that

$$Same_L(kind(ev), \textbf{state when } ev, (\textbf{state when } ev) \setminus L)$$

We will use this, call it (**), in the proofs of (b), (c), and (d) in this remaining case.

For part (b), we proceed as in the proof of Lemma 2, letting p be the local predecessor of e.

$$\textbf{state when } e = \textbf{Trans}(kind(p), val(p), \textbf{state when } p)$$

By the induction hypothesis, the state $S_1 = (\textbf{state when } p) \setminus L$ is independent of atom a and we must show that state $S_2 = (\textbf{state when } e) \setminus L$ is independent of atom a. If $kind(p) \neq k$, then, by (**),

$$\textbf{state when } e = \textbf{Trans}(kind(p), val(p), S_1) \textbf{ mod } L$$

Thus,

$$S_2 = \textbf{Trans}(kind(p), val(p), S_1) \setminus L$$

The function $\textbf{Trans}(kind(p))$ is independent of $a \in ats$ by assumption (12). The expressions $val(p)$ and S_1, are independent of a by parts (c) and (b) of the induction hypothesis. And, $\lambda s.s[L := \textbf{nil}]$ is trivially independent of a. Using independent-Application we deduce that $S_2 \| a$ finishing part (b) in the case $kind(p) \neq k$.

If $kind(p) = k$, then we use constraints (3) and (7) to deduce that

$$\textbf{state when } e = (\textbf{state when } p)[ptr := ptr \textbf{ when } p + 1]$$

Thus

$$S_2 = S_1[ptr := ptr \textbf{ when } p + 1]$$

By the induction hypothesis, S_1 is independent of a. The integer ptr **when** $p + 1$ is independent of a by Lemma 1, and this is enough to deduce that $S_2 \| a$ finishing the proof of (b).

To prove (c) we must show that $(val(e) \| a)$. If e is a receive event, then we have already seen that $(val(e) \| a)$ follows from part (d) of the induction hypothesis. Otherwise, e is an internal event, and therefore it is not a nonce request, so, $kind(e) \neq k$. In this case,

$$val(e) = \textbf{IVal}(loc(e), x, n, \textbf{state when } e)$$

for some number n, and action name x, and again we use (**) to conclude that

$$val(e) = \textbf{IVal}(loc(e), x, n, (\textbf{state when } e) \setminus L)$$

Since we have already proved in part (b) that $(((\textbf{state when } e) \setminus L) \| a)$ and all the other parts of the expression for $val(e)$ are independent of a by our assumptions, this finishes the proof of part (c).

To prove part (d), we must show that if e' is a receive event on some link l with some tag tg and $sender(e') = e \neq sender(n)$ then $(val(e') \| a)$. If $kind(e) \neq k$, then we proceed as in the proof of Corollary 2:

$$\langle l, tg, val(e') \rangle \in \textbf{Send}(kind(e), val(e), \textbf{state when } e)$$

Then, using (**),

$$\langle l, tg, val(e')\rangle \in \textbf{Send}(kind(e), val(e), (\textbf{state when } e) \setminus L)$$

By what we have already proved, we have $(val(e)\|a)$ and $(\textbf{state when } e) \setminus L\|a$, and this is enough to finish the proof of part (d) when $kind(e) \neq k$.

Finally, when $kind(e) = k$ then, by constraint (8), we have

$$l = l_i \ \wedge \ tg = nonce$$

Then, by constraint (4) and the fact that L **when** $e = ats$,

$$(ptr \textbf{ when } e < len(ats) \ \wedge \ val(e') = ats[ptr \textbf{ when } e]$$

Thus it is enough to prove $ats[ptr \textbf{ when } e]\|a$, and by rule "independentBase" this reduces to showing $ats[ptr \textbf{ when } e] \neq a$. Since $a = ats[ptr \textbf{ when } sender(n)]$ and, by assumption, *ats* has no repeats, it is enough to show that

$$ptr \textbf{ when } e \neq ptr \textbf{ when } sender(n)$$

In the case we are considering, both e and $sender(n)$ have location s_i and kind k. Since events at a single location are totally ordered, and $e \neq sender(n)$, we must have $e <_{loc} sender(n)$ or $sender(n) <_{loc} e$. By constraint 3,

$$\forall e.\ kind(e) = k \Rightarrow ptr \textbf{ when } e < ptr \textbf{ after } e$$

It follows easily, by induction on $<_{loc}$, from the constraints (3) and (6), that for any events e_1 and e_2 with location s_i ,

$$kind(e_1) = k \ \wedge \ e_1 <_{loc} e_2 \ \Rightarrow \ ptr \textbf{ when } e_1 < ptr \textbf{ when } e_2$$

With this, the proof of part (d) is easily finished. □

We have described the nonce server and given the proof of the nonce property at the level of detail needed for the formal proof in NuPrl.

Before we present an application of our nonces, it will be convenient to introduce some general specification concepts that are expressed as properties of event structures and events.

4.3 Event Classes

An *event class* (of type T) is simply a partial function from events to values of type T. In type theory this is represented by a function of type $E \rightarrow T + Unit$. If X is an event class, then we write $E(X)$ for the set of events in the domain of X and say that events in $E(X)$ are *in* the class X. For such an event $e \in E(X)$, $X(e)$ is the value (of type T) assigned to e by class X.

Thus, class X partitions the event structure and for those events e that are in $E(X)$ assigns a value of type T. We use the notational shorthand $e \in X(v)$ to mean that event $e \in E(X)$ and $X(e) = v$. If X is a class of a product type $T_1 \times \cdots \times T_k$

then we extend this shorthand so that $e \in X(v_1, \ldots, v_k)$ means that $e \in E(X)$ and $X(e) = \langle v_1, \ldots, v_k \rangle$.

Note that an event e may be in any number of different event classes. If Y is event class of type T', and f is a function from $T \to T'$, then the notation $X(v) \Rightarrow Y(f(v))$ means

$$\forall e.\, e \in X(v) \Rightarrow e \in Y(f(v))$$

This says that any event in class X is also in class Y and its value in class Y is a function of its value in class X.

We say that a class X is *locally defined* if it is a function of the local properties of its input. That means that whether or not $e \in E(X)$ is a function of the kind and value of e, and the state when e occurs, and if $e \in E(X)$ the value $X(e)$ is also a function of the kind and value of e, and state when e occurs. For example, the class Val_k for which $E(\mathit{Val}_k) = \{e \mid \mathit{kind}(e) = k\}$ and $\mathit{Val}_k(e) = val(e)$ is locally defined.

With this terminology we can state a more general version of the nonce interface property as a corollary of the fundamental nonce theorem.

Corollary 3. *For any nonce $n \in E(\mathit{Nonce}_i)$ and any event $e \in E(X)$ where X is a locally defined class X,*

$$(\mathit{loc}(e) \neq s_i \ \wedge \ n \not\leq e \Rightarrow X(e) \| \mathit{Nonce}_i(n)$$

5 Sequential Laws

Definition 5. *The* sequential law $A;_R B$ *is the conjunction of the two constraints*

$$\forall e, a.\, e \in A(a) \Rightarrow \exists e', b.\, e < e' \ \wedge \ e' \in B(b) \ \wedge \ R(a, b)$$

and

$$\forall e', b.\, e' \in B(b) \Rightarrow \exists e, a.\, e < e' \ \wedge \ e \in A(a) \ \wedge \ R(a, b)$$

This law says that if an event of either class A or class B occurs, then an event of the other class is guaranteed, with a properly related value and in the proper order.

A particularly useful special case of this scheme is when class A has type T_1 and class B has type $T_1 \times T_2$ and relation R holds only between values of the form a and $\langle a, b \rangle$. In this case we will simply write $A; B$ for the conjunction of

$$\forall e, a.\, e \in A(a) \Rightarrow \exists e', b.\, e < e' \ \wedge \ e' \in B(a, b)$$

and

$$\forall e', a, b.\, e' \in B(a, b) \Rightarrow \exists e.\, e < e' \ \wedge \ e \in A(a)$$

Let us write the first of these constraints as

$$e \in A(a) \hookrightarrow e' \in B(a, b)$$

and the second constraint as

$$e \in A(a) \hookleftarrow e' \in B(a, b)$$

Definition 6. *The* iterated sequential law $A; B; C$ *is the conjunction of the laws* $A; B$ *and* $B; C$

If $A; B; C$ holds then:

$$e \in A(a) \hookrightarrow e' \in B(a, b) \hookrightarrow e'' \in C(a, b, c)$$

and

$$e \in A(a) \hookleftarrow e' \in B(a, b) \hookrightarrow e'' \in C(a, b, c)$$

and

$$e \in A(a) \hookleftarrow e' \in B(a, b) \hookleftarrow e'' \in C(a, b, c)$$

Thus the occurrence of an event in any of the classes A, B, or C implies the existence of properly ordered events in the other two classes, with properly related values.

5.1 Model of Digital Signature

To model cryptographic services we introduce other virtual servers. Agents communicate with a virtual server over secure links, so a virtual server acts like a trusted third party. But it is only a *virtual* agent used to prove that the logical properties of the cryptographic service are consistent. When we generate code, we will replace sends to and receives from the virtual server with calls to the real cryptographic system it is modeling. So no real trusted third party will be needed.

To model different crypto-systems we vary the state and interface of the server to correspond with the services the system provides. For example, a model of a digital signature service is similar to the nonce server, in that it assumes a list of atoms unique to the server. The state of the server consists of a three-column table tab with the atoms $a_1, a_2, a_3, \ldots$ in the the first column and a pointer ptr into the table. When the server receives a message $sign(v)$ on the link from agent j when the pointer has value n, then it searches for a row of the form $\langle a_m, v, j\rangle$ for which $m < n$. If it finds one it sends the response a_m to agent j. Otherwise it sets row n to $\langle a_n, v, j\rangle$, increments the pointer, and sends the "signature" a_n to agent j in response. When the server receives a message $verify(sig, v, j)$ from an agent i, it searches for a row of the form $\langle a_n, v, j\rangle$ for which $n < ptr$ and $a_n = sig$. If it finds one it sends the response "true" to agent i and otherwise sends "false".

In terms of this model we can define event classes *Sign* and *Verify*. A signing event $e \in$ *Sign*(B, v, sig) will be an event where agent B receives signature sig in response to signing value v. A verify event $e \in$ *Verify*(B, v, sig) is associated with the same tuple $\langle B, v, sig\rangle$, and represents the successful verification of the signature by the agent at *loc*(e). The classes *Sign* and *Verify* are related by the law:

$$e \in \textit{Sign}(signer, v, sig) \hookleftarrow e' \in \textit{Verify}(signer, v, sig)$$

which says that a successful verify event must be preceded by a matching signing event.

Also, those signing events $e \in$ *Sign*(B, v, sig) that are the first signing by B of the value v are also nonces and satisfy the general nonce property in corollary 3.

6 Authentication Example

Consider the following challenge-response protocol:

$$\begin{array}{lcl} \text{Initiator}(A) & \text{Network} & \text{Responder}(B) \\ \text{Send-Nonce} & \xrightarrow{A, B, n} & \text{Receive}(n') \text{ from } A' \\ & & sig = \text{Sign}_B(\langle A', n' \rangle) \\ \text{Receive}(sig) \text{ from } B & \xleftarrow{B, A', sig} & \text{Send} \\ \text{Verify}(B, sig, \langle A, n \rangle) & & \end{array}$$

Analysis of such protocols must take into account the possibility that each participant may be engaging, simultaneously, in several instances (in either role) of the protocol. For this reason, we will associate with the first event in each initiator or responder sequence an identifier k (which could simply be a number). Our proof is also simplified if we associate each event with its location.

An instance of the initiator role will thus be a sequence of three events in event classes I_1, I_2, I_3 with associated values of the form:

$$I_1(A, k, n, B); I_2(A, k, n, B, sig); I_3(A, k, n, B, sig)$$

An instance of the responder role is a sequence of three events in classes R_1, R_2, R_3 with values of the form:

$$R_1(B, k, A, n); R_2(B, k, A, n, sig); R_3(B, k, A, n, sig)$$

Events in each of these classes will also be members of one of the classes Nonce_A, *Sign*,*Verify*, *Send*, or *Receive*. The classes *Send* and *Receive* represent send and receive events to and from an unreliable network, and both have associated values of the form $\langle from, to, msg \rangle$.

Using our notation $X(v) \Rightarrow Y(f(v))$ to describe the relations between classes, we have:

$$I_2(A, k, n, B) \Rightarrow \mathit{Nonce}_A(n) \ \wedge \ \mathit{Send}(A, B, n) \tag{15}$$
$$I_2(A, k, n, B, sig) \Rightarrow \mathit{Receive}(B, A, sig) \tag{16}$$
$$I_3(A, k, n, B, sig) \Rightarrow \mathit{Verify}(B, \langle A, n \rangle, sig) \tag{17}$$
$$R_1(B, k, A, n) \Rightarrow \mathit{Receive}(A, B, n) \tag{18}$$
$$R_2(B, k, A, n, sig) \Rightarrow \mathit{Sign}(B, \langle A, n \rangle, sig) \tag{19}$$
$$R_3(B, k, A, n, sig) \Rightarrow \mathit{Send}(B, A, sig). \tag{20}$$

An attacker does whatever it can, but the honest participants in this protocol will be *law abiding*. A law abiding initiator satisfies the laws

$$I_1 \hookleftarrow I_2 \ \wedge \ I_2 \hookleftarrow I_3$$

This is because the initiator verifies only signatures received by events in class I_2 and receives events in class I_2 only after initiating the protocol with an event in class I_1.

A law abiding responder satisfies the sequential law $R_1; R_2; R_3$, thus if an event in any of these classes occurs then matching events in all three classes occur. The responder B must also satisfy

$$e \in \mathit{Sign}(B, \langle A, n\rangle, sig) \Rightarrow \exists k.\, e \in R_2(B, k, A, n, sig)$$

This means that a law-abiding responder B never signs any pair except as part of some instance of its responder role in this protocol. (To make this assumption more reasonable, the protocol could be ammended so that the responder signs, and the intiator verifies, a tuple of the form $\langle xxx, A, n\rangle$ where xxx is a constant "header" unique to instances of this authentication protocol.)

6.1 Correctness Proof

We are now ready to derive the correctness of the authentication protocol. The correctness statement is that any event $e_3 \in I_3(A, k, n, B, sig)$ where A and B are law abiding, implies that a matching conversation between A and B has occurred. In this case, a matching conversation between A and B is a sequence of four events $a_1 < b_1 < b_2 < a_2$ where a_1 and a_2 are a send and receive at A and b_1 and b_2 are a receive and send at B, and values of the receive events match the values of the preceding send event in the sequence.

Since A is law abiding, we have events $e_1 < e_2 < e_3$ with

$$\begin{aligned} a_1 &\in I_1(A, k, n, B)\\ a_2 &\in I_2(A, k, n, B, sig)\\ a_3 &\in I_3(A, k, n, B, sig) \end{aligned}$$

Then we also have

$$\begin{aligned} a_1 &\in \mathit{Nonce}_A(n)\\ a_1 &\in \mathit{Send}(A, B, n)\\ a_2 &\in \mathit{Receive}(B, A, sig)\\ a_3 &\in \mathit{Verify}(B, \langle A, n\rangle, sig) \end{aligned}$$

The last of these implies that there exists event $b_2 \in \mathit{Sign}(B, \langle A, n\rangle, sig)$ with $b_2 < a_3$. Since B is law abiding, for some k', $b_2 \in R_2(B, k', A, n, sig)$, and from the sequential law $R_1; R_2; R_3$ we derive the existence of events b_1 and b_3 with $b_1 < b_2 < b_3$ and

$$\begin{aligned} b_1 &\in R_1(B, k', A, n)\\ b_2 &\in R_2(B, k', A, n, sig)\\ b_3 &\in R_3(B, k', A, n, sig) \end{aligned}$$

Then we also have

$$\begin{aligned} b_1 &\in \mathit{Receive}(A, B, n)\\ b_3 &\in \mathit{Send}(B, A, sig) \end{aligned}$$

Thus the events a_1, b_1, b_3, a_3 form a matching conversation between A and B, provided that we can show

$$a_1 < b_1 < b_3 < a_3$$

From $b_1 \in \mathit{Receive}(A, B, n)$ and $a_1 \in \mathit{Nonce}_A(n)$ we deduce that $a_1 < b_1$. We already have $b_1 < b_3$ from the sequential law for B, so it remains to show that $b_3 < a_3$. This follows from the fact that $b_2 \in \mathrm{Sign}(B, \langle A, n\rangle, sig)$ is the only signing event at location B that includes nonce n, and the signature sig is therefore also a nonce, which is not sent by law abiding B until event b_3. Since $a_3 \in \mathit{Receive}(B, A, sig)$ we must have $b_3 < a_3$.

7 Related Work

We have already discussed the relation between our work, the use of atoms in logic by Gabbay and Pitts [11], and the work on Choiceless Polynomial Time by Blass, Gurevich and Shelah [5].

Related work on formal proofs of security is vast (see [14] for one survey), but we call attention to the Protocol Composition Logic (PCL) [8]. PCL is a logic for proving security properties of protocols that use public and symmetric key cryptography. The logic contains axioms and inference rules that allow assertions to be proved by reasoning about the actions of the honest parties in the protocol, and these assertions hold even in the face of additional actions by a malicious adversary.

This corresponds exactly with our method of deriving properties that hold in all event structures consistent with a program. We believe that the PCL proof system can be embedded into our event logic by a so-called *deep embedding*. To do this all of the axioms and rules of PCL would have to be proved for their corresponding embedding in event logic. PCL has axioms that express properties of nonces in terms of predicates "Has", "Fresh", 'FirstSend", and temporal ordering $<$. These properties should follow from the fundamental nonce property proved in Theorem 1.

In [7], Datta et al. show that *ideal functionalities* for many fundamental security mechanisms cannot be realized by a "real" protocol. The notion of an ideal functionality for a cryptographic primitive corresponds with our concept of a virtual server that models it. The impossibility result shows that most of these virtual servers will require secure links to more than one agent, thus making them into trusted third parties and not "real". As we have discussed, this does not invalidate the use of virtual agents as a means for proving the logical properties of cryptographic primitives from an abstract model where they are not "built in".

Strand spaces [20] provide extra structure to a symbolic model. The general properties that are deducible from this structure are mostly already captured by the axioms and rules of PCL, so an embedding of PCL into our framework would extend to an embedding of strand space models as well.

Paulson's inductive approach to proving security [17, 18] and its elaboration by Millen and Rueß [15] provide organizing principles for carrying out proofs of secrecy. A protocol-independent *secrecy theorem* proved by induction on a well-founded ordering on traces reduces proofs of secrecy to a "local" condition on the honest agents called *discreetness*. We expect to be able to translate all these concepts and theorems

into our framework. By doing this we will strengthen the results by proving them for a more general adversary model and we will be able to generate running code from the message automata extracted from applications of these theorems.

8 Conclusions and Future Work

We have shown that the concepts of atoms and independence are well-behaved and can be added to most logical systems. This makes it possible to combine a general model of computation with a simple model of acquisition of secret information. We have shown how to model cryptographic primitives with automata exchanging atoms. This allows us to specify and prove their logical properties in single unifying framework.

Clearly, what we have accomplished is only a beginning. As Datta, Mitchell, et.al. write:

> Our eventual goal is to assign to each protocol component, refinement and transformation a logical formula, expressing its meaning, and formally tied to it by a semantical relation ...

The results in this paper are all implemented in Nuprl, so the foundation for a more ambitious project, such as embedding PCL and automating proofs in the PCL system, is already in place.

Acknowledgments

We would like to thank: Stuart F. Allen for many hours of discussions about the rules for atoms and independence; Robbert van Renesse for the virtual server idea; David Guaspari for editing and reorganizing several drafts of this report; Bob Constable for enthusiastic technical and financial support for this work and collaboration since 2003; the IAI (Information Assurance Institute) for sponsoring this research, and NSF grant CNS-0614790.

References

[1] Allen, S.F.: A non-type-theoretic definition of martin-löf's types. In: Proceedings of Second IEEE Symposium on Logic in Computer Science, pp. 215–224 (1987)
[2] Allen, S.F.: An Abstract Semantics for Atoms in Nuprl. Cornell Tech. Report TR2006-2032 (2006)
[3] Bickford, M.: Event systems. Nuprl Math Library Book webpage (2003)
[4] Bickford, M., Constable, R.L.: A logic of events. Technical Report TR2003-1893, Cornell University (2003)
[5] Blass, A., Gurevich, Y., Shelah, S.: Choiceless polynomial time. Pure and Applied Logic 100, 141–187 (1999)
[6] Constable, R.L.: Types in logic, mathematics and programming. In: Buss, S.R. (ed.) Handbook of Proof Theory, pp. 683–786. Elsevier Science B.V, Amsterdam (1998)
[7] Datta, A., Derek, A., Mitchell, J., Ramanathan, A., Scedrov, A.: Games and the impossibility of realizable ideal functionality. In: Proceedings of 16th IEEE Computer Security Foundations Workshop, pp. 360–379 (March 2006)

[8] Datta, A., Derek, A., Mitchell, J.C., Roy, A.: Protocol composition logic (pcl). In: Plotkin Festschrift, G.D. (ed.) Electronic Notes in Theoretical Computer Science (to appear, 2007)
[9] Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
[10] Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Multiset rewriting and the complexity of bounded security protocols (2002)
[11] Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing 13, 341–363 (2002)
[12] Goldreich, O.: Foundations of Cryptography. Basic Tools, vol. 1. Cambridge University Press, Cambridge (2001)
[13] Martin-Löf, P.: Constructive mathematics and computer programming. In: 6th International Congress for Logic, Methodology, and Philosophy of Science, North Holland, Amsterdam (1982)
[14] Meadows, C.A.: Formal verification of cryptographic protocols: A survey. In: Safavi-Naini, R., Pieprzyk, J.P. (eds.) ASIACRYPT 1994. LNCS, vol. 917, pp. 133–152. Springer, Heidelberg (1995)
[15] Millen, J.K., Rueß, H.: Protocol-independent secrecy. In: IEEE Symposium on Security and Privacy, pp. 110–209 (2000)
[16] Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
[17] Paulson,: Proving security protocols correct. In: LICS: IEEE Symposium on Logic in Computer Science (1999)
[18] Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6(1-2), 85–128 (1998)
[19] Pfenning, F., Schürmann, C.: Twelf — a meta-logical framework for deductive systems, pp. 202–206 (1999)
[20] Thayer, F.J., Herzog, J.C., Guttman, J.D.: Strand spaces: Proving security protocols correct. Journal of Computer Security 7(1) (1999)
[21] Troelstra, A.: Choice Sequences. Clarendon Press, Oxford (1977)

A Definition of Independence

In Martin-Löf's semantics for constructive type theory [13], propositions are types, and a type is a *partial equivalence relation* on terms, so to define new propositions we must

1. Define the type expressions that represent the new propositions
2. Define when two of these type expressions represent the same type
3. Define the members of the type and the equivalence relation on the members
4. Show that inference rules about the type are valid

These tasks are done in the *metatheory* where the primitive objects are *terms* and there is a primitive *evaluates to* relation on terms. Validity of inference rules is defined in terms of the semantics of *sequents* for which details can be found in [6] and in the article by Allen [1].

Definition 7. *The expression* $(x{:}T\|a)$ *is a type whenever* $a \in Atom$, $T \in Type$[1], *and* $x \in T$.

[1] In Nuprl. there is no type $Type$ of all types; instead there is a cumulative hierarchy of *universes*. In this paper, we use the symbol $Type$ for an arbitrary universe.

$(x{:}T\|a) = (x'{:}T'\|a')$ *(i.e. they represent the same type) if* $a = a' \in Atom$, $T = T' \in Type$, *and* $x = x' \in T$.

Only terms with the special canonical form $Axiom$ *can be members of the type* $(x{:}T\|a)$, *and, any two members are equivalent.*

To complete the definition of the type $(x{:}T\|a)$ we have to define when it has a member, i.e. when it is *true*.

Definition 8. *The proposition* $(x{:}T\|a)$ *is true if and only if* a *evaluates to* $tok(b)$ *for some name* b, *and there exists a term* y *in type* T *such that* $x = y \in T$ *and* y *does not mention name* b.

This completes our definition of the new primitive proposition $(x{:}T\|a)$. The next lemma shows that the definition is well formed and that the rule "independentEquality" is valid.

Lemma 3. *If* $(x{:}T\|a) = (x'{:}T'\|a')$ *then* $(x{:}T\|a) \Leftrightarrow (x'{:}T'\|a')$.

Proof. Assume that $a = a' \in Atom$, $T = T' \in Type$, $x = x' \in T$, $y \in T$, $x = y \in T$, a evaluates to $tok(b)$, and y does not mention b. We must show that a' evaluates to $tok(b')$ for some name b', and that there exists a $y' \in T'$ for which $x' = y' \in T'$ and y' does not mention b'. Atoms have unique canonical forms, so since $a = a' \in Atom$, a' must evaluate to $tok(b)$, so we can take b'=b. Then we can take y' = y, since y does not mention b and $x = y \in T$ and $x = x' \in T$ and $T = T' \in Type$ imply that $x' = y \in T'$. □

The validity, according to Martin-Löf's semantics, of the other inference rules in table 1 is easily proved. As examples, we will give the proofs for the application and absurdity rules.

Lemma 4 (independentApplication). *If* $(x{:}A\|a)$ *and* $(f{:}(x : A \rightarrow B[x])\|a)$ *then* $(f(x){:}B[x]\|a)$

Proof. If a evaluates to $tok(b)$ and $f = f' \in (x : A \rightarrow B[x])$ and f' does not mention b, and if $x = x' \in A$ and x' does not mention b, then the term $f'(x')$ does not mention b and by the definition of the dependent function type,
$x : A \rightarrow B[x]$, we have $f(x) = f'(x') \in B[x]$. □

Lemma 5 (independentAbsurdity). $(a{:}Atom\|a) \Rightarrow False$

Proof. If a evaluates to $tok(b)$ and $y = a \in Atom$ and y does not mention b, then we have y=$tok(b) \in Atom$, by computation, and $y = tok(c) \in Atom$, by the permutation rule for names. Thus $tok(b) = tok(c) \in Atom$, and this implies $False$. □

B Inherence Application Principle Not Constructive

We can show that the inherence application principle

$$(f(x){:}B[x/v] \gg a) \Rightarrow$$
$$((f{:}(v : A \rightarrow B[v]) \gg a) \vee (x{:}A \gg a))$$

is not constructively valid if we admit functions that are not recursive but can still be considered constructible. These functions are *random choice sequences* analogous to Brouwer's free choice sequences (see [21] for more on choice sequences). For the purpose of this section we need only sequences of outcomes from the space $\{0, 1\}$ where 0 and 1 each have probability $1/2$, but the theory of random sequences we use generalizes to sequences of outcomes from any finite probability space.

We add to the computation system a table $T(a, n)$ whose rows are indexed by atoms a and whose columns are indexed by natural numbers n. At all times, table T is finite, so that it contains only finitely many triples $\langle a, n, T(a, n)\rangle$, but as time progresses more triples are added as follows. Whenever the computation system evaluates a term of the form $rand(a, n)$, it computes the canonical forms of a and n and returns the value of $T(a, n)$ if it is in the table. Otherwise it "flips a fair coin" and fills in the value of $T(a, n)$ according to whether it gets a head or tail. Thus the term $rand(a) = \lambda n.\ rand(a, n)$ is a member of the type $\Omega = \mathbb{N} \to \mathbb{N}_2$. The concept of an open subset of Ω of measure one can be easily defined (with no need to define topology or measure in general). It is consistent to add the following **axiom**:

> If C is an open set of measure one and C is independent of atom a, then there exists n such that the basic open set determined by $rand(a, 0), \ldots, rand(a, n)$ is contained in C.

This axiom asserts that the random sequences in the rows of table T are mutually generic random sequences. In fact, for our argument that the inherence application principle is not constructive we need only one generic random sequence $rand(a)$.

For any function f in the type $\mathbb{N} \to \mathbb{N}$, define the sequence r_f by $r_f(n) = rand(a, n)$, if $f(n)$ is different from $f(0), \ldots, f(n-1)$, and $r_f(n) = 0$, otherwise. If we think of f as enumerating a recursively enumerable set A, then if x is enumerated into A at time n, $r_f(n) = rand(a, n)$, and if nothing is added to A at time n, then $r_f(n) = 0$.

Now we claim that if f itself is independent of atom a, then atom a is inherent in the sequence r_f if and only if the set $A = range(f)$ is infinite. This is the same as asserting that r_f is independent of atom a if and only if A is finite. One direction of this assertion is clear. If A is finite then $r_f(n)$ will be 0 for all but finitely many n, so there is a finite term that does not mention atom a that is equal to sequence r_f. If A is infinite, then the set of sequences g for which there is an n where $g(n) = 0$ and $r_f(n) = 1$ is an open set of measure one (because, since f and hence A is independent of a, $rand(a, n) = 1$ for infinitely many of the n for which something is added to A at time n). If r_f is independent of atom a, then this open set is independent of a, and hence the axiom implies that $rand(a)$ meets the set, but this is impossible since $r_f(n) = 1$ implies $rand(a, n) = 1$. Thus r_f is not independent of a if A is infinite.

Now given two functions, f and g, both independent of a, we can form the pointwise maximum, $max(r_f, r_g)$. Atom a is inherent in this sequence if and only if the union of $range(f)$ and $range(g)$ is infinite. If the inherence application principle were constructive, there would be a constructive procedure to decide whether atom a was inherent in r_f or in r_g, but this would provide a constructive procedure to decide, given two r.e. sets whose union is infinite, which of them is infinite, and there is clearly no constructive procedure to decide this.

Combining Domain-Specific and Foundational Logics to Verify Complete Software Systems

Xinyu Feng[1], Zhong Shao[2], Yu Guo[3], and Yuan Dong[4]

[1] Toyota Technological Institute at Chicago
[2] Yale University
[3] University of Science and Technology of China
[4] Tsinghua University

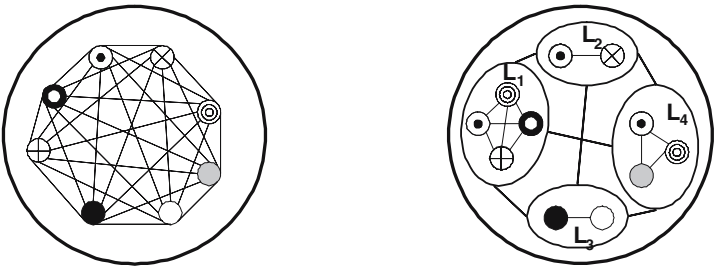
Abstract. A major challenge for verifying *complete* software systems is their complexity. A complete software system consists of program modules that use many language features and span different abstraction levels (*e.g.,* user code and run-time system code). It is extremely difficult to use one verification system (*e.g.,* type system or Hoare-style program logic) to support all these features and abstraction levels. In our previous work, we have developed a new methodology to solve this problem. We apply specialized "domain-specific" verification systems to verify individual program modules and then link the modules in a foundational open logical framework to compose the verified complete software package. In this paper, we show how this new methodology is applied to verify a software package containing implementations of preemptive threads and a set of synchronization primitives. Our experience shows that domain-specific verification systems can greatly simplify the verification process of low-level software, and new techniques for combining domain-specific and foundational logics are critical for the successful verification of complete software systems.

1 Introduction

It is difficult to verify complete software systems because they use many different language features and span different abstraction levels. As an example, our ongoing project to verify a simplified operating system kernel exposes such challenges. The kernel has a simple bootloader, kernel-level threads and a thread scheduler, synchronization primitives, hardware interrupt handlers, and a simplified keyboard driver. Although it has only around 1,300 lines of x86 assembly code, it already uses features such as dynamic code loading, thread scheduling and context switching, concurrency, hardware interrupts, device drivers and I/O. *How do we verify safety and correctness of the whole system with machine-checkable proofs?*

Verifying the whole system in a single type system or program logic is impractical because, as Fig. 1 (a) shows, such a verification system needs to consider all possible interactions among these different features, many of which are even at different abstraction levels. The resulting logic, if exists, would be very complex and difficult to use. Fortunately, in reality, programmers seem to never use all the features at the same time. Instead, only limited combinations of features—at certain abstraction level—are used in individual program modules. It would be much simpler to design and use specialized "domain specific" logics to verify individual program modules, as shown in

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 54–69, 2008.



(a). The One-for-All Logic (b). Domain-Specific Logics

Fig. 1. Using Domain Specific Logics to Verify Modules

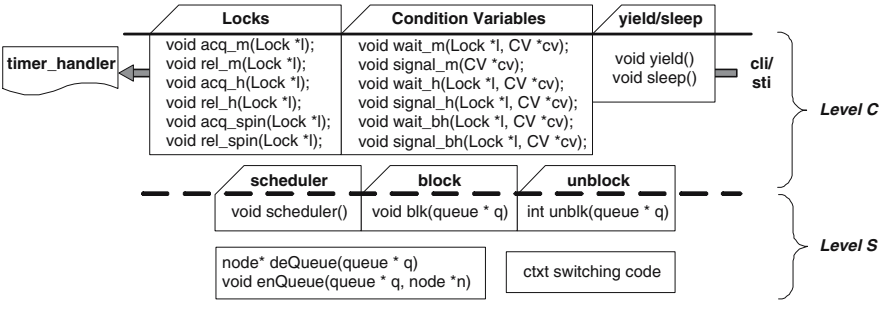


Fig. 2. The Thread Implementations

Fig. 1 (b). To allow interactions of modules and build a complete system, we need to also support the interoperability of different logics.

In our previous work [6], we proposed a new methodology for modular verification and linking of low-level software systems. Different type systems and program logics can be used to verify different program modules. These verification systems (called *domain-specific systems* or *foreign systems*) are then embedded into a foundational open framework, OCAP, where interoperability is supported. Verified program modules in a foreign system can be reused in the framework, given a sound embedding of the foreign system. Safety properties of the whole program can be derived from verified modules.

In this paper, we show how to use this methodology to verify the implementations of preemptive threads and a set of synchronization primitives extracted from our OS kernel. As shown in Fig. 2, the modules include thread scheduling, blocking and unblocking, a timer interrupt handler, implementations of locks and condition variables, and yield and sleep primitives. To verify them, we develop OCAP-x86, which adapts the OCAP framework for the x86 architecture and extends it with the support of hardware interrupts. We then introduce four domain-specific program logics, each designed for a specific set of program modules that share the same abstraction level. These logics are all embedded into OCAP-x86. The soundness of the embedding and the interoperability of foreign logics are formally proved in the Coq proof assistant [4]. Program modules are verified in "domain-specific" foreign logics with machine-checkable proofs.

This verification effort depends critically on our previous work on the OCAP theories [6] and specialized program logics for reasoning about interrupts [7] and runtime stacks [9]. In this paper, instead of developing new theories or logics, we present a case study of our methodology and report our experience in applying it to fully verify a relatively complex software package. Specifically, our experience shows that our verification project can benefit from this new methodology in many different ways.

1. Implementations of thread primitives are at a lower abstraction level than synchronization primitives. It is very natural to verify them using different logics.
2. Using specialized logics, we can hide irrelevant details about the environment from the specification of program modules, and greatly simplify the inference rules for reasoning about programs. These details are added back when the foreign logics are embedded into OCAP-x86. This improves the modularity and the reusability of the verified primitives.
3. By abstracting away the thread implementation details in the logics used to verify synchronization primitives, we avoid supporting first-class code pointers in these logics, even though the underlying thread primitives treat program counters saved in thread control blocks (TCBs) as normal data.
4. Using specialized logics is an effective way to ban some instructions in certain specific modules. For instance, the "iret" instruction is only used to "return" from interrupt handlers to non-handler code, and should not be used outside handlers.

In the rest of this paper, we first show in Sect. 2 the challenges to verify the package and give an overview of our solutions. We present our modeling of the x86 assembly language in Sect. 3 and the OCAP-x86 framework in Sect. 4. We then explain in detail the verification of thread implementations in Sect. 5 and synchronization primitives in Sect. 6. We give more details of our Coq implementations in Sect. 7, and discuss about related work and conclude in Sect. 8.

2 Background and Challenges

The program modules in Fig. 2 are extracted from our simplified OS kernel (for uniprocessor platforms only) which is implemented in 16-bit x86 assembly code and works in real mode. The scheduler saves the execution context of the current thread into the ready queue, picks another one from the queue, and switches to the execution context of the new thread. block takes a pointer to a block queue as argument, puts the current thread into the block queue, and switches the control to a thread in the ready queue. unblock also takes a pointer to a block queue as argument; it moves a thread from the block queue to the ready queue but does not do context switching. Execution of threads may be interrupted by hardware interrupts. When the interrupt request comes, the control is transferred to the interrupt handler. To implement preemptive threads, the handler calls scheduler before it returns. Threads can also yield the control voluntarily by calling scheduler. block and unblock are used to implement synchronization primitives.

There are many challenges to fully verify these program modules. Our previous work has solved problems related to specific language features, such as control abstractions

[9] and preemptive threads with hardware interrupts [7]. However, to verify all the modules as part of a whole system, we still need to address the following problems.

How to verify thread implementations? The code implementing synchronization primitives is concurrent because it may be run by different threads and be preempted by interrupt handlers. It needs to be verified using a concurrent program logic. However, it is difficult to use the same logic to verify scheduler, block and unblock. Concurrent program logics, *e.g.,* rely-guarantee reasoning [11] or Concurrent Separation Logic [15], usually assume built-in concurrency and abstract away schedulers. The scheduler and the block primitives, on the other hand, switch the execution contexts of threads. They are at a lower abstraction level than threads.

Another problem is that the thread primitives manipulate threads' execution contexts containing program counters pointing to the code where threads resume execution. These code pointers are stored in memory, thus can be read and updated as normal data. To verify thread implementations using a concurrent program logic, we need to support "first-class" code pointers, which is difficult in Hoare-style reasoning and the solutions are usually heavyweight [13].

How to enforce local invariants? Program invariants play an important role in verification. Knowing that the invariants always hold, we can use them without putting them in specifications, which simplifies verification steps and improves modularity. Although there is no meaningful system-wide invariants other than the most basic safety requirements, more refined invariants are preserved locally in different program modules. How do we take advantage of these local invariants and ensure that they are preserved in corresponding modules? For instance, to avoid race conditions, the implementation of scheduler, block and unblock needs to disable interrupts. Therefore the potential interrupts by interrupt handlers should never be a concern when verifying these subroutines. Another example is that thread code, including implementations of synchronization primitives, assumes thread queues are well-formed. Also the well-formedness is trivially preserved because the thread code never directly accesses these queues other than through thread primitives. Since these invariants are only preserved locally in individual modules, it is difficult to enforce them globally in the program logic.

How to ban certain instructions in specific modules? As mentioned before, interrupts are not concerned in thread primitives, assuming they are always disabled. However, this invariant cannot be preserved if the sti instruction is executed, which enables interrupts. Actually thread implementations should only use a subset of instructions and are not supposed to execute any interrupt-related ones. Similarly, the iret instruction in x86 is the last instruction of interrupt handlers. It returns control to the non-handler code. This instruction cannot be used interchangeably with the normal ret instruction for functions, and should not be used outside of interrupt handlers. We need to have a way to ban these instructions in specific scenarios.

To solve these problems, we verify the package following the open-framework-based methodology. As shown in Fig. 2, we split the code into two layers: the upper level C (for "Concurrent") and the low level S (for "Sequential"). Code at Level C is concurrent; it handles interrupts explicitly and implements interrupt handlers but abstracts away the implementation of threads. Code at Level S is sequential (always executed with

interrupt disabled); functions that need to know the concrete representations of thread control blocks (TCBs) and thread queues are implemented at Level S.

We use a Hoare-style program logic for sequential assembly code to verify Level S. Knowing that interrupts are always disabled, specifications for Level S do not need to mention the fact at all. To ban interrupt-related instructions at this level, we simply exclude any rules for these instructions in the specialized logic, so code containing these instructions cannot be verified. It is also interesting to note that we do not need to support first-class code pointers in this level to verify context switching code, because program counters saved in TCBs are treated as normal data by thread primitives. Although they point to code at Level C, Level C is not verified *within* this logic.

We hide the implementation of threads from Level C. To do this, we design an abstract machine for the higher level code, which treats scheduler, block, and unblock as abstract primitive instructions. Thread queues are abstract algebraic structures (*e.g.,* sets) instead of data structures in memory. They are inaccessible by normal instructions except scheduler, block and unblock. Various program logics are used to verify modules at this level. In these logics, we do not need to specify the well-formedness of thread queues in memory. This not only improves the modularity and reusability of verified modules, but avoids the support of first-class code pointers because code pointers in the thread queues are no longer first-class in this abstract machine.

3 The Machine

Figure 3 shows our modeling of a subset of 16-bit x86 machines. The machine configuration contains the code $\mathbb{C}$, the program state $\mathbb{S}$ and the program counter pc. The code maps code labels to commands. The program state contains the memory, the register file, the flag register and a special register isr.

There are several simplifications in our modeling of the x86 architecture. The code $\mathbb{C}$ is not part of the heap and is read-only. The von Neumann architecture can be supported following Cai *et al.* [3]. We do not model the full PIC (programmable interrupt controller). Instead, we assume only the timer interrupt is allowed and all other interrupts are masked. The isr register, which is now degenerated into a binary value, records

$$
\begin{array}{rrcl}
(\textit{World}) & \mathbb{W} & ::= & (\mathbb{C},\mathbb{S},\mathtt{pc}) \\
(\textit{Code}) & \mathbb{C} & ::= & \{\mathtt{f}_1 \rightsquigarrow \mathtt{c}_1,\ldots,\mathtt{f}_n \rightsquigarrow \mathtt{c}_n\} \\
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{M},\mathbb{R},\mathbb{F},\mathsf{isr}) \\
(\textit{Mem}) & \mathbb{M} & ::= & \{\mathtt{l}_1 \rightsquigarrow \mathtt{w}_1,\ldots,\mathtt{l}_n \rightsquigarrow \mathtt{w}_n\} \\
(\textit{RegFile}) & \mathbb{R} & ::= & \{\mathsf{ax} \rightsquigarrow \mathtt{w}_1,\ldots,\mathsf{sp} \rightsquigarrow \mathtt{w}_8\} \\
(\textit{FlagReg}) & \mathbb{F} & ::= & \{\mathsf{if} \rightsquigarrow \mathtt{b}_1,\mathsf{zf} \rightsquigarrow \mathtt{b}_2\}
\end{array}
$$

$$
\begin{array}{rrcl}
(\textit{Word}) & \mathtt{w} & ::= & i\ (\textit{nat numbers}) \\
(\textit{Labels}) & \mathtt{l},\mathtt{f},\mathtt{pc} & ::= & i\ (\textit{nat numbers}) \\
(\textit{Boolean}) & \mathtt{b},\mathsf{isr} & ::= & \mathtt{tt} \mid \mathtt{ff} \\
(\textit{Addr}) & \mathtt{d} & ::= & i \mid i(\mathtt{r}) \\
(\textit{Oprand}) & \mathtt{o} & ::= & i \mid \mathtt{r}
\end{array}
$$

$$
\begin{array}{rrcl}
(\textit{Reg}) & \mathtt{r} & ::= & \mathsf{ax} \mid \mathsf{bx} \mid \mathsf{cx} \mid \mathsf{dx} \mid \mathsf{bp} \mid \mathsf{di} \mid \mathsf{si} \mid \mathsf{sp} \\
(\textit{Instr}) & \mathtt{i} & ::= & \mathsf{add}\ \mathtt{r},\mathtt{o} \mid \mathsf{sub}\ \mathtt{r},\mathtt{o} \mid \mathsf{mov}\ \mathtt{r},\mathtt{o} \mid \mathsf{mov}\ \mathtt{r},\mathtt{d} \mid \mathsf{mov}\ \mathtt{d},\mathtt{r} \mid \mathsf{cmp}\ \mathtt{r},\mathtt{o} \mid \mathsf{jne}\ \mathtt{f} \\
 & & & \mid \mathsf{push}\ \mathtt{o} \mid \mathsf{push}\ \mathtt{d} \mid \mathsf{pushf} \mid \mathsf{pop}\ \mathtt{o} \mid \mathsf{pop}\ \mathtt{d} \mid \mathsf{popf} \mid \mathsf{sti} \mid \mathsf{cli} \mid \mathsf{eoi} \mid \mathsf{call}\ \mathtt{f} \\
(\textit{Command}) & \mathtt{c} & ::= & \mathtt{i} \mid \mathsf{jmp}\ \mathtt{f} \mid \mathsf{ret} \mid \mathsf{iret}
\end{array}
$$

Fig. 3. The Machine

$$
\begin{array}{rrcl}
(\mathit{CHSpec}) & \Psi & \in & \mathcal{P}(\mathit{Labels} * \mathit{OCdSpec}) \\
(\mathit{OCdSpec}) & \pi & ::= & \langle \rho, \mathcal{L}, \theta \rangle \quad \in \mathit{LangID} * (\Sigma X : \mathsf{Type}.\, X) \\
(\mathit{LangID}) & \rho & ::= & n\ (\mathit{nat\ nums}) \\
(\mathit{LangTy}) & \mathcal{L} & ::= & (\mathit{CiC\ terms}) \quad \in \mathsf{Type} \\
(\mathit{CdSpec}) & \theta & ::= & (\mathit{CiC\ terms}) \quad \in \mathcal{L} \\
(\mathit{Assert}) & \mathsf{a} & \in & \mathit{CHSpec} \to \mathit{State} \to \mathsf{Prop} \\
(\mathit{Interp}) & [\![\_]\!]_{\mathcal{L}} & \in & \mathcal{L} \to \mathit{Assert} \\
(\mathit{LangDict}) & \mathcal{D} & \in & \mathit{LangID} \rightharpoonup \Sigma X : \mathsf{Type}.\, (X \to \mathit{Assert})
\end{array}
$$

Fig. 4. Specification Constructs of OCAP-x86

whether the timer interrupt is currently being handled. If it is set, new interrupt requests will not be processed. The bit needs to be cleared before the interrupt handler returns. The instruction eoi clears isr. It is a shorthand for the real x86 instruction "out dx, al", with the precondition that ax and dx both contain value 0x20.

The if flag records whether interrupts are enabled. It can be set by sti and cleared by cli. At each program point, the control is transferred to the entry point of the interrupt handler if there is an incoming request and the interrupt is enabled. Otherwise the next instruction at pc is executed. Instead of modeling the source of the interrupt, we use a non-deterministic semantics and assume the interrupt may come at any moment. Formula (1) formalizes the condition under which an interrupt request will be handled:

$$\mathsf{enable\_itr}(\mathbb{S}) \stackrel{\mathbf{def}}{=} (\mathbb{S}.\mathbb{F}(\mathsf{if}) = \mathtt{tt}) \wedge (\mathbb{S}.\mathsf{isr} = \mathtt{ff})\,. \tag{1}$$

Because we support only the timer interrupt, we assume there is a global entry point ih_entry of the interrupt handler. When an interrupt request is handled, the encoding of $\mathbb{F}$ and pc is pushed onto the stack[1]; if is cleared and isr is set; and pc is set to ih_entry. Semantics of other instructions are standard. The formal definition of operational semantics can be find in our Coq implementation [8] (the file `Operational_Semantics.v`).

4 The OCAP-x86 Framework

OCAP-x86 adapts OCAP [6] for our x86 machine and extends it to support interrupts. It is developed in Coq [4], which implements CiC [17] and supports mechanized theorem proving in Higher-Order Logic. In Coq, the universe of logical propositions is Prop. Coq can also be used as a meta-language so that domain-specific logics can be defined using inductive definitions. These inductively defined types in Coq are in universe Type.

In OCAP-x86, the code heap specification Ψ for $\mathbb{C}$ associates code labels with generic specifications π, which encapsulate concrete specifications in domain-specific logics into a uniform format. Each π is a triple containing an identifier ρ of a domain-specific logic, the CiC meta-type $\mathcal{L}$ of its specification language, and a concrete specification θ in $\mathcal{L}$. In OCAP-x86, specifications from domain-specific logics are mapped to a foundational form a, which is a logical predicate over Ψ and program states $\mathbb{S}$. For each

[1] Some arbitrary value is also pushed onto stack as the "cs" register.

$\mathcal{L}$, the mapping is done by the interpretation $[\![\_]\!]_{\mathcal{L}}$. The dictionary $\mathcal{D}$ of specification languages then maps language identifiers ρ to the specification language's meta-type and the corresponding interpretation. We allow one specification language $\mathcal{L}$ to have multiple interpretations, each with a unique ρ in $\mathcal{D}$.

$$\frac{\begin{array}{l}(\mathsf{ih\_entry}, \pi_0) \in \Psi \quad \forall \Psi', \mathbb{S}.\ \mathtt{a}\ \Psi'\ \mathbb{S} \wedge \Psi \subseteq \Psi' \\ \qquad \rightarrow \exists \mathtt{f}', \mathbb{S}', \pi'.\ (\mathtt{f}' = \mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})) \wedge (\mathbb{S}' = \mathsf{next}_{(\mathtt{f},\iota)}(\mathbb{S})) \\ \qquad\qquad \wedge (\mathtt{f}', \pi') \in \Psi \wedge [\![\pi']\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}' \\ \qquad\qquad \wedge (\mathsf{enable\_itr}(\mathbb{S}) \rightarrow \exists \mathbb{S}' = \mathsf{next\_itr}(\mathbb{S}, \mathtt{f}).\ [\![\pi_0]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}')\end{array}}{\mathcal{D}; \Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \iota}\ (\textsc{ins})$$

The INS rule just shown is a schema in OCAP-x86 for all instructions. Informally, the judgment $\mathcal{D}; \Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \iota$ says if the precondition $\mathtt{a}$ holds, it is safe to execute the instruction at the label $\mathtt{f}$ and the resulting state satisfies the post-condition in Ψ (which is also the precondition of the following computation). $\mathcal{D}$ is used to interpret specifications in Ψ. $\mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})$ and $\mathsf{next}_{(\mathtt{f},\iota)}(\mathbb{S})$ define the new pc and state after executing ι at $\mathtt{f}$ with the initial state $\mathbb{S}$. The specification at $\mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})$ in Ψ is used as the post-condition. In addition to the sequential execution, we also need to consider the interrupt if $\mathsf{enable\_itr}(\mathbb{S})$ holds. In this case, the next state needs to satisfy the precondition π_0 for the interrupt handler. $\mathsf{next\_itr}(\mathbb{S})$ is the next state after the interrupt comes at state $\mathbb{S}$. $\mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})$, $\mathsf{next}_{(\mathtt{f},\iota)}(\mathbb{S})$ and $\mathsf{next\_itr}(\mathbb{S})$ are part of the operational semantics and the details are omitted here. $[\![\pi]\!]_{\mathcal{D}}$ is the interpretation of π in $\mathcal{D}$, which is defined as:

$$[\![\langle \rho, \mathcal{L}, \theta \rangle]\!]_{\mathcal{D}} \stackrel{\text{def}}{=} \lambda \Psi, \mathbb{S}.\ \exists [\![\_]\!]_{\mathcal{L}}.\ \mathcal{D}(\rho) = (\mathcal{L}, [\![\_]\!]_{\mathcal{L}}) \wedge [\![\theta]\!]_{\mathcal{L}}\ \Psi\ \mathbb{S}. \tag{2}$$

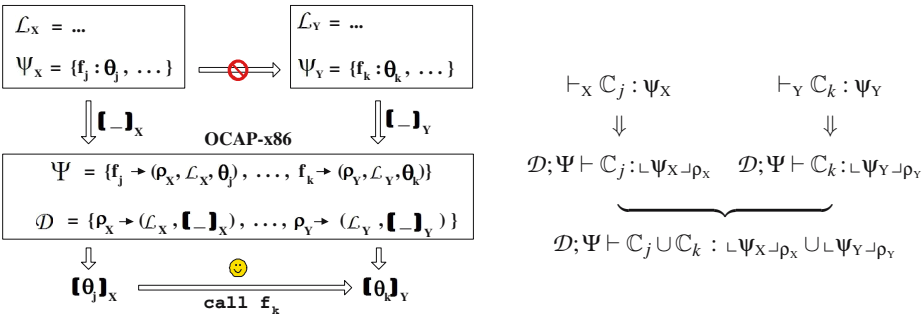
The LINK rule below links separately verified modules $\mathbb{C}_1$ and $\mathbb{C}_2$.

$$\frac{\mathcal{D}_1; \Psi_1 \vdash \mathbb{C}_1 : \Psi_1' \quad \mathcal{D}_2; \Psi_2 \vdash \mathbb{C}_2 : \Psi_2' \quad \mathcal{D}_1 \# \mathcal{D}_2 \quad \mathbb{C}_1 \# \mathbb{C}_2}{\mathcal{D}_1 \cup \mathcal{D}_2; \Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1' \cup \Psi_2'}\ (\textsc{link})$$

We use $f \# f'$ to mean the partial functions f and f' maps the same value to the same image. The judgment $\mathcal{D}; \Psi \vdash \mathbb{C} : \Psi'$ means the module $\mathbb{C}$ is verified with imported interface Ψ and exported interface Ψ'. The complete set of OCAP-x86 rules is presented in the companion technical report [8].

How to use OCAP-x86. Inference rules of OCAP-x86 are designed for generality with minimum constraints instead of for ease of use. Actually the rules are not intended to be used directly by program verifiers. Instead, domain-specific logics should be designed above the foundational framework to verify specific modules. The specifications and rules in the domain-specific logics can be customized and simplified, given the specific local invariants preserved in the modules.

Figure 5(a) shows two domain-specific logics with different specification languages $\mathcal{L}_X$ and $\mathcal{L}_Y$. Each has its own customized rules. To prove the customization is sound, we encode the local invariants in the interpretations $[\![\_]\!]_X$ and $[\![\_]\!]_Y$, and show premises of the customized rules imply premises of the corresponding OCAP-x86 rules, thus these domain-specific rules are admissible in the foundational framework.



(a) Embedding (b) Linking

Fig. 5. Interfacing and Linking Modules

Suppose we have modules verified in $\mathcal{L}_X$ and $\mathcal{L}_Y$, with specifications ψ_X and ψ_Y respectively. They cannot be interfaced directly because ψ_X and ψ_Y are specified in different languages. We map the concrete specifications (*e.g.,* θ_j and θ_k) into the generic triples π in the Ψ at the OCAP-x86 level, where ρ_X and ρ_Y are language IDs assigned to $\mathcal{L}_X$ and $\mathcal{L}_Y$. Their interpretations are put in $\mathcal{D}$. To interface modules, we match in OCAP-x86 the interpreted specifications, which are now assertions a. Since the interpretation usually encodes program invariants, matching the interfaces at this level guarantees the local invariants are compatible at boundaries of modules.

Given the soundness of the embeddings, the modules verified in $\mathcal{L}_X$ and $\mathcal{L}_Y$ can be reused in OCAP-x86 without redoing the proof (see Fig. 5(b), where $\llcorner\psi\lrcorner_\rho$ maps ψ to Ψ in OCAP-x86). Then they can be linked using the LINK rule. The soundness of OCAP-x86 guarantees that the complete system after linking never gets stuck. Also the interpretation of specifications in Ψ holds when the corresponding program points are reached by jmp or call instructions.

Theorem 1 (Soundness of OCAP-x86).
If $\mathcal{D}; \Psi \vdash (\mathbb{C}, \mathbb{S}, \mathtt{pc})$, then, for all n, there exist $\mathbb{S}'$ and $\mathtt{pc}'$ such that $(\mathbb{C}, \mathbb{S}, \mathtt{pc}) \longmapsto^n (\mathbb{C}, \mathbb{S}', \mathtt{pc}')$, and there exists $\Psi' \supseteq \Psi$ such that the following are true:

1. if $\mathbb{C}(\mathtt{pc}') = \mathsf{jmp}\ \mathtt{f}$, there exists π such that $(\mathtt{f}, \pi) \in \Psi$ and $[\![\pi]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}'$;
2. if $\mathbb{C}(\mathtt{pc}') = \mathsf{call}\ \mathtt{f}$, there exist π and $\mathbb{S}''$ such that $(\mathtt{f}, \pi) \in \Psi$, $\mathbb{S}'' = \mathsf{next}_{(\mathtt{pc}', \mathsf{call}\ \mathtt{f})}(\mathbb{S}')$, and $[\![\pi]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}''$;
3. if $\mathsf{enable\_itr}(\mathbb{S}')$, there exist π_0 and $\mathbb{S}''$ such that $(\mathsf{ih\_entry}, \pi_0) \in \Psi$, $\mathbb{S}'' = \mathsf{next\_itr}(\mathbb{S}', \mathtt{pc}')$, and $[\![\pi_0]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}''$.

Here $\mathcal{D}; \Psi \vdash \mathbb{W}$ means the whole system $\mathbb{W}$ is well-formed with respect to Ψ (see the TR [8] for its definition); $\mathbb{W} \longmapsto^n \mathbb{W}'$ means an n-step transition from $\mathbb{W}$ to $\mathbb{W}'$.

5 Verifying Thread Implementations

The implementation of threads contains the three thread primitives at Level S in Fig. 2. The scheduler uses the simple FIFO scheduling policy. The thread queues are

$$
\begin{array}{rrcl}
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{M}, \mathbb{R}, \mathbb{F}, \cancel{\mathbb{isr}}) \qquad\qquad (\textit{FlagReg})\ \mathbb{F} ::= \{\cancel{\mathtt{if} \rightsquigarrow \mathtt{b}_1}, \mathtt{zf} \rightsquigarrow \mathtt{b}_2\} \\
(\textit{Instr}) & \iota & ::= & \mathtt{add\ r,o} \mid \mathtt{sub\ r,o} \mid \mathtt{mov\ r,o} \mid \mathtt{mov\ r,d} \mid \mathtt{mov\ d,r} \mid \mathtt{cmp\ r,o} \mid \mathtt{jne\ f} \\
 & & & \mid \mathtt{push\ o} \mid \mathtt{push\ d} \mid \cancel{\mathtt{pushf}} \mid \mathtt{pop\ o} \mid \mathtt{pop\ d} \mid \cancel{\mathtt{popf}} \mid \cancel{\mathtt{sti}} \mid \cancel{\mathtt{cli}} \mid \cancel{\mathtt{eoi}} \mid \mathtt{call\ f} \\
(\textit{Command}) & \mathtt{c} & ::= & \iota \mid \mathtt{jmp\ f} \mid \mathtt{ret} \mid \cancel{\mathtt{iret}}
\end{array}
$$

Fig. 6. A Sequential Subset of x86 Machine

$$
\begin{array}{rrcl}
(\textit{StPred}) & \mathtt{p} & \in & \textit{State} \to \mathsf{Prop} \\
(\textit{Guarantee}) & \mathtt{g} & \in & \textit{State} \to \textit{State} \to \mathsf{Prop}
\end{array}
\qquad
\begin{array}{rrcl}
(\textit{LangTy}) & \mathcal{L}_{\mathrm{SCAP}} & ::= & \textit{StPred} * \textit{Guarantee} \\
(\textit{CdSpec}) & \theta & ::= & (\mathtt{p}, \mathtt{g}) \\
(\textit{LocalSpec}) & \psi & ::= & \{\mathtt{f}_1 \rightsquigarrow \theta_1, \ldots, \mathtt{f}_n \rightsquigarrow \theta_n\}
\end{array}
$$

Fig. 7. SCAP-x86 Specifications

implemented as doubly-linked lists containing thread control blocks (TCBs). Each TCB contains the status of the thread (Ready, Idle or Blocked) and the value of the sp register when the thread is preempted or blocked. Note that we do not need to store pc in TCB because, by the calling convention, pc is at the top of the stack pointed by sp.

As explained in Sect. 1, the TCB of the running thread is put into the ready queue if it is preempted or it calls scheduler voluntarily. The scheduler sets its status to Ready unless it is an idle thread. When block is called, the current thread is put into the corresponding block queue with the status Blocked. An idle thread with the status Idle would never be blocked. This guarantees the ready queue is not empty when block is called.

To avoid race conditions, code at this level needs to disable interrupts. As a result, the full x86 machine can be simplified into a sequential subset shown in Fig. 6, where anything related to interrupts are removed. It is the machine model in the programmers' mental picture while they are writing and verifying code at Level S. Correspondingly, we use a sequential program logic, SCAP-x86, to verify the thread implementations. SCAP-x86 adapts the SCAP logic [9] to the x86 architectures. The specification constructs are shown in Fig. 7. The specification θ is instantiated using a pair $(\mathtt{p}, \mathtt{g})$. p is a logical predicate in Coq specifying the precondition over the program state; g is a predicate over a pair of states, which specifies the state transition from the current program point to the end of the current function. The concrete specification ψ of code heaps in SCAP-x86 maps code labels to θ. Note that we do not really implement SCAP-x86 on the sequential machine in Fig. 6. The states specified in p and g are still x86 program states defined in Fig. 3. However, p and g can leave if and isr unspecified, as if they are specifying the simpler states in Fig. 6. To exclude these interrupts-related instructions, SCAP-x86 simply does not have inference rules for them, therefore any modules using these instructions cannot be verified in SCAP-x86.

Specifications of thread primitives. Figure 8 shows our definitions of abstract thread queues and TCBs. The set of all threads is a partial mapping T from thread IDs to abstract TCBs. Each tcb contains the thread status and the values of sp and pc. Note that, in physical memory, pc is stored on top of stack instead of in the TCB. We put it in the abstract tcb for convenience of specifications. The abstract queue Q is just a set of thread IDs. B is a partial mapping from block queue IDs to queues for blocked threads.

$$
\begin{array}{rrcl}
(\mathit{ThrdSet}) & \mathsf{T} & ::= & \{\mathtt{t}_1 \rightsquigarrow \mathsf{tcb}_1, \ldots, \mathtt{t}_n \rightsquigarrow \mathsf{tcb}_n\} \\
(\mathit{BlkQSet}) & \mathsf{B} & ::= & \{b_1 \rightsquigarrow \mathsf{Q}_1, \ldots, b_n \rightsquigarrow \mathsf{Q}_n\} \\
(\mathit{ThrdQ}) & \mathsf{Q} & ::= & \{\mathtt{t}_1, \ldots, \mathtt{t}_n\}
\end{array}
\qquad
\begin{array}{rrcl}
(\mathit{Status}) & \mathtt{s} & ::= & \mathsf{Ready} \mid \mathsf{Idle} \mid \mathsf{Blocked} \\
(\mathit{TCB}) & \mathsf{tcb} & ::= & (\mathtt{s}, \mathtt{w}, \mathtt{pc}) \\
(\mathit{ThrdID}) & \mathtt{t} & ::= & n \text{ (nat nums and } n > 0) \\
(\mathit{BQID}) & b & ::= & n \text{ (nat nums and } n > 0)
\end{array}
$$

Fig. 8. Abstract Queues and TCBs

$$
\begin{array}{l}
\mathsf{sch\_pre\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathtt{m}) \stackrel{\mathbf{def}}{=} \\
\quad \lambda(\mathbb{M}, \mathbb{R}, \mathbb{F}, \mathsf{isr}).\ (\mathsf{T}(\mathtt{t}) = \mathsf{tcb}) \wedge (\mathbb{R}(\mathsf{sp}) = \mathtt{l}) \\
\qquad \wedge (\mathsf{embed\_curr\_TCB}(\mathtt{t}, \mathsf{tcb}) * \mathsf{embed\_RdyQ}(\mathsf{T}, \mathsf{Q}) * (\mathtt{l} \overset{2}{\mapsto} \mathtt{pc}) * \mathsf{stkSpace}(\mathtt{l}, \mathsf{sch\_stk}) * \mathtt{m})\ \mathbb{M} \\
\mathsf{sch\_post\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{l}_0, \mathtt{pc}_0, \mathtt{m}) \stackrel{\mathbf{def}}{=} \\
\quad \lambda(\mathbb{M}, \mathbb{R}, \mathbb{F}, \mathsf{isr}).\ (\mathsf{T}(\mathtt{t}) = \mathsf{tcb}) \wedge (\mathsf{tcb} = (\_, \mathtt{l}, \_)) \wedge (\mathbb{R}(\mathsf{sp}) = \mathtt{l}) \\
\qquad \wedge (\mathsf{embed\_curr\_TCB}(\mathtt{t}, \mathsf{tcb}) * \mathsf{embed\_RdyQ}(\mathsf{T}, \mathsf{Q}) * (\mathtt{l}_0 \overset{2}{\mapsto} \mathtt{pc}_0) * \mathsf{stkSpace}(\mathtt{l}_0, \mathsf{sch\_stk}) * \mathtt{m})\ \mathbb{M} \\
\mathsf{sch\_pre} \stackrel{\mathbf{def}}{=} \lambda\mathbb{S}.\ \exists \mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}.\ \mathsf{sch\_pre\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathsf{true})\ \mathbb{S} \\
\mathsf{sch\_grt} \stackrel{\mathbf{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \forall \mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathtt{m}.\ \mathsf{sch\_pre\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathtt{m})\ \mathbb{S} \\
\qquad \rightarrow \exists \mathsf{T}', \mathtt{t}', \mathsf{tcb}', \mathsf{Q}', \mathtt{l}'.\ (\mathsf{T}' = \mathsf{T}\{\mathtt{t} \rightsquigarrow (\mathsf{tcb}.\mathtt{s}, \mathtt{l}, \mathtt{pc})\}) \wedge (\mathsf{Q} \cup \{\mathtt{t}\} = \mathsf{Q}' \cup \{\mathtt{t}'\}) \\
\qquad\qquad \wedge \mathsf{sch\_post\_aux}(\mathsf{T}', \mathtt{t}', \mathsf{tcb}', \mathsf{Q}', \mathtt{l}', \mathtt{l}, \mathtt{pc}, \mathtt{m})\ \mathbb{S}'
\end{array}
$$

Fig. 9. The Specification of scheduler

The specification of scheduler is $(\mathsf{sch\_pre}, \mathsf{sch\_grt})$, as shown in Fig. 9. sch_pre is the precondition. It says that T maps the ID of the current thread to its TCB, which has a concrete representation in memory ($\mathsf{embed\_curr\_TCB}(\mathtt{t}, \mathsf{tcb})$); the ready queue Q is embedded in memory, storing TCBs in T ($\mathsf{embed\_RdyQ}(\mathsf{T}, \mathsf{Q})$); a pc is at the top of stack pointed by sp ($\mathbb{R}(\mathsf{sp}) = \mathtt{l}$ and $\mathtt{l} \overset{2}{\mapsto} \mathtt{pc}$); and there is sufficient stack space for the scheduler function ($\mathsf{stkSpace}(\mathtt{l}, \mathsf{sch\_stk})$, where sch_stk is the size of the stack space for scheduler). Here we use separating conjunction in Separation Logic [18] to mean the current TCB, the ready queue and the stack are in different portions of memory. We use $\mathtt{l} \overset{2}{\mapsto} \mathtt{w}$ to mean w is stored at l and l+1 (*i.e.,* a 16-bit integer). Also the specification is polymorphic over the part of memory untouched by scheduler, represented by the memory predicate m. We omit the concrete definitions of the embedding of TCBs and thread queues here. Interested readers can refer to our Coq implementation for details [8].

The auxiliary definition sch_post_aux specifies the post-condition. Here the parameters (T, t, tcb *etc.*) refer to the state immediately before scheduler returns, except $\mathtt{l}_0$ and $\mathtt{pc}_0$, which refer to the value of sp and pc at the beginning of the function. The guarantee sch_grt requires that the pre- and post-conditions hold over states at the beginning and at the end, respectively. It also relates the auxiliary variables in sch_post_aux with those in sch_pre_aux, and ensures that the calling thread of scheduler is added into the ready queue Q; its sp and pc is saved in TCB and is put in T; other TCBs in T are preserved; and the memory specified by m is unchanged. Specifications for block and unblock are in similar forms and not shown here.

SCAP-x86 and the embedding in OCAP-x86. Inference rules in SCAP-x86 are similar to those in the original SCAP for a MIPS-like architecture [9]. Unlike rules in OCAP-x86,

where interrupts have to be considered at every step of verification, the SCAP-x86 rules only consider the sequential execution of instructions.

To embed SCAP-x86 into OCAP-x86, we first define the interpretation below.

$$[\![(\mathtt{p},\mathtt{g})]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}} \stackrel{\mathbf{def}}{=} \lambda\Psi,\mathbb{S}.\ \mathtt{p}\,\mathbb{S} \wedge (\mathbb{S}.\mathbb{F}(\mathsf{if}) = \mathtt{ff}) \wedge \exists n.\ \mathsf{WFST}(n,\rho,\mathtt{g},\mathbb{S},\mathcal{D},\Psi)$$

$$\mathsf{WFST}(0,\rho,\mathtt{g},\mathbb{S},\mathcal{D},\Psi) \stackrel{\mathbf{def}}{=} \forall\mathbb{S}'.\ \mathtt{g}\,\mathbb{S}\,\mathbb{S}' \to \exists \mathsf{ra},\pi.\ (\mathsf{ra} = \mathsf{Ret\_Addr}(\mathbb{S}')) \wedge (\mathsf{ra},\pi) \in \Psi \wedge [\![\pi]\!]_{\mathcal{D}}\,\Psi\,\mathsf{next}_{\mathsf{ret}}(\mathbb{S}')$$

$$\mathsf{WFST}(n{+}1,\rho,\mathtt{g},\mathbb{S},\mathcal{D},\Psi) \stackrel{\mathbf{def}}{=} \forall\mathbb{S}'.\ \mathtt{g}\,\mathbb{S}\,\mathbb{S}' \to \exists \mathsf{ra},\mathtt{p}',\mathtt{g}',\mathbb{S}''.\ (\mathsf{ra} = \mathsf{Ret\_Addr}(\mathbb{S}')) \wedge (\mathsf{ra},\langle\rho,\mathcal{L}_{\mathrm{SCAP}},(\mathtt{p}',\mathtt{g}')\rangle) \in \Psi \wedge \mathbb{S}'' = \mathsf{next}_{\mathsf{ret}}(\mathbb{S}') \wedge \mathtt{p}'\,\mathbb{S}'' \wedge \mathsf{WFST}(n,\rho,\mathtt{g}',\mathbb{S}'',\mathcal{D},\Psi)$$

The interpretation $[\![\_]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}}$ maps $(\mathtt{p},\mathtt{g})$ to an assertion $\mathtt{a}$ in OCAP-x86. It takes the id ρ assigned to SCAP-x86 and a dictionary $\mathcal{D}$ as open parameters. $\mathcal{D}$ is used to interpret foreign modules that will interact with modules verified in SCAP-x86; it is decided at the time of linking. The invariant about interrupts is added back, although it is omitted in $\mathtt{p}$. For any $\mathtt{p}$ and $\mathtt{g}$, we have $\forall\Psi,\mathbb{S}.\ [\![(\mathtt{p},\mathtt{g})]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}}\,\Psi\,\mathbb{S} \to \neg\mathsf{enable\_itr}(\mathbb{S})$. The predicate WFST says that, starting from $\mathbb{S}$, every time a function returns (*i.e.,* its $\mathtt{g}$ is fulfilled), the return address is a valid code pointer in Ψ and its specification is satisfied after ret. Here $\mathsf{Ret\_Addr}(\mathbb{S})$ gets the return address saved on the stack of $\mathbb{S}$.

The embedding of SCAP-x86 into OCAP-x86 is sound. The soundness theorem is given below. Given a program module $\mathbb{C}$ verified in SCAP-x86, we can apply the theorem to convert it into a verified package in OCAP-x86. When the code at the Level S in Fig. 2 is verified, (*i.e.,* when $\psi \vdash \mathbb{C} : \psi'$ is derived in SCAP-x86), we do not need any knowledge of Level C and the foreign logics in which Level C is verified.

Theorem 2 (Soundness of the Embedding of SCAP-x86). Suppose ρ is the id assigned to SCAP-x86. For all $\mathcal{D}$, let $\mathcal{D}' = \mathcal{D}\{\rho \rightsquigarrow (\mathcal{L}_{\mathrm{SCAP}}, [\![\_]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}})\}$. If $\psi \vdash \mathbb{C} : \psi'$ in SCAP-x86, we have $\mathcal{D}'; \llcorner\psi\lrcorner_\rho \vdash \mathbb{C} : \llcorner\psi'\lrcorner_\rho$ in OCAP-x86, where $\llcorner\psi\lrcorner_\rho \stackrel{\mathbf{def}}{=} \{(\mathtt{f},(\rho,\mathcal{L}_{\mathrm{SCAP}},\theta)) \mid \psi(\mathtt{f}) = \theta\}$.

6 Verifying Synchronization Primitives and Interrupt Handlers

The code at Level C implements synchronization primitives and a timer interrupt handler. It handles interrupts explicitly and is executed by preemptive threads. It also calls the thread primitives at Level S, therefore thread queues need to be well-formed in memory. However, since all accesses to thread queues are made through these thread primitives, thread queues would always be well-formed if we treat thread primitives as atomic operations at this level. Thus we can hide this invariant from specifications, as we hide interrupts in SCAP-x86. Based on this idea, we develop an abstract interrupt machine (AIM-x86), which treats thread primitives as atomic instructions. Representations of queues and TCBs (*e.g.,* $\mathsf{embed\_RdyQ}(\mathtt{T},\mathtt{Q})$ and $\mathsf{embed\_curr\_TCB}(\mathtt{t},\mathtt{tcb})$) are abstracted away. Abstract representations in Fig. 8 are used instead.

Figure 10 shows the AIM-x86 abstract machine, a variation of our AIM machine [7]. We extend the x86 world with the thread ID of the current thread, a mapping of thread

$$(\textit{World})\ \ \mathbb{W} ::= (\mathbb{C}, \mathbb{S}, \mathtt{pc}, \mathtt{t}, \mathsf{T}, \mathsf{Q_r}, \mathsf{B})$$
$$(\textit{Instr})\ \ \iota ::= \ldots \mid \mathsf{scheduler} \mid \mathsf{block} \mid \mathsf{unblock} \mid \ldots$$

Fig. 10. The AIM-x86 Machine

IDs to their TCBs, a ready queue and a set of block queues, which are all defined in Fig. 8. Then the three primitive instructions are added. The rest part of the machine are the same as in Fig. 3. Below we show the operational semantics for scheduler:

$$\frac{\begin{array}{c}\mathbb{C}(\mathtt{pc}) = \mathsf{scheduler} \quad \mathbb{F}(\mathsf{if}) = \mathbb{F}'(\mathsf{if}) = \mathtt{ff} \quad \mathsf{isr} = \mathtt{ff} \quad \mathsf{T}(\mathtt{t}) = (\mathtt{s}, \_, \_) \quad \mathbb{R}(\mathsf{sp}) = \mathtt{l}{+}2 \\ \mathsf{T}' = \mathsf{T}\{\mathtt{t} \rightsquigarrow (\mathtt{s}, \mathtt{l}, \mathtt{pc}{+}1)\} \quad \mathsf{Q_r} \cup \{\mathtt{t}\} = \mathsf{Q}'_\mathsf{r} \cup \{\mathtt{t}'\} \quad \mathsf{T}(\mathtt{t}') = (\_, \mathtt{l}', \mathtt{pc}') \quad \mathbb{R}'(\mathsf{sp}) = \mathtt{l}'{+}2 \\ \mathbb{M} = \mathbb{M}_1 \uplus \mathbb{M}_2 \quad (\mathtt{l} \overset{2}{\mapsto} \_ * \mathtt{l}{-}2 \overset{2}{\mapsto} \_ * \cdots * \mathtt{l}{-}\mathsf{sch\_stk} \overset{2}{\mapsto} \_)\, \mathbb{M}_2 \\ \mathbb{M}' = \mathbb{M}_1 \uplus \mathbb{M}'_2 \quad (\mathtt{l} \overset{2}{\mapsto} \_ * \mathtt{l}{-}2 \overset{2}{\mapsto} \_ * \cdots * \mathtt{l}{-}\mathsf{sch\_stk} \overset{2}{\mapsto} \_)\, \mathbb{M}'_2\end{array}}{(\mathbb{C}, (\mathbb{M}, \mathbb{R}, \mathbb{F}, \mathsf{isr}), \mathtt{pc}, \mathtt{t}, \mathsf{T}, \mathsf{Q_r}, \mathsf{B}) \longmapsto (\mathbb{C}, (\mathbb{M}', \mathbb{R}', \mathbb{F}', \mathsf{isr}), \mathtt{pc}', \mathtt{t}', \mathsf{T}', \mathsf{Q}'_\mathsf{r}, \mathsf{B})}\ (\textsc{sch})$$

Before executing scheduler, the interrupt needs to be disabled. The scheduler saves sp (actually sp-2) and the return address (pc$+1$) into the TCB of the current thread. A thread $\mathtt{t}'$ is picked to run and the control is transferred to $\mathtt{pc}'$, which is loaded from the TCB of $\mathtt{t}'$. The memory $\mathbb{M}$ can be split into $\mathbb{M}_1$ and $\mathbb{M}_2$. $\mathbb{M}_2$ is the stack used by scheduler. $\mathbb{M}_1$ is untouched by scheduler. We can see the semantics is consistent with sch_grt in Fig. 9, modulo the fact that the SCH rule specifies states before calling scheduler and after the return of it, while sch_grt specifies states after the call and before the return. Since pc's in thread queues are not accessible by normal instructions, they are not first-class data and we do not need to guarantee their validity in our program logic, which is challenging in Hoare-style reasoning [13].

Domain-Specific Logics for Level C. We have presented a program logic for AIM [7] in our previous work. Here we use variations of the logic to verify our code. These variations are not developed for the extended instruction sets in the abstract AIM-x86 machine; instead, scheduler, block and unblock are treated as aliases for the real x86 instructions "call `scheduler`", "call `block`" and "call `unblock`". Similar to the sequential subset of x86 in Fig. 6, AIM-x86 is simply a conceptual machine model only for the design of our domain-specific program logics; it is never implemented.

We use customized program logics, SCAP-Rdy, SCAP-Idle and SCAP-Itr, to verify normal threads, the idle thread and interrupt handlers respectively. In SCAP-Rdy, we exclude the inference rules for "iret" and "eoi", which are supposed to be used only in interrupt handlers. The set of SCAP-Idle rules is a strict subset of SCAP-Rdy excluding the rule for "block", so that we can guarantee there is at least one thread in the ready queue when the running thread is blocked. SCAP-Itr has rules for "iret" and "eoi". In the interpretation for SCAP-Itr, we distinguish the outermost level function (the interrupt handler) and functions called by the interrupt handler, since the former returns using "iret". The rules for the common instructions supported in the three logics are the same, therefore normal functions need to be verified only once and can be reused.

Embedding in OCAP-x86. We need to define interpretations for the logics to embed them in OCAP-x86. Since the well-formedness of thread queues (*e.g.,* embed_RdyQ(T, Q))

is assumed as invariants in the logics and is unspecified in program specifications, we need to add it back in our interpretation so that the preconditions of thread primitives at Level S (*e.g.,* sch_pre in Fig. 9) are satisfied when they are called from Level C.

The soundness of the embedding is similar to Theorem 2, which says inference rules in the customized logics can be viewed as lemmas in the OCAP-x86 frame work based on the interpretation. The following lemma says the rule for scheduler in SCAP-Rdy is sound, given any specification of scheduler compatible with $(\mathsf{sch\_pre}, \mathsf{sch\_grt})$.

Lemma 3 (Interfacing with scheduler**).** Suppose ρ_r is the id for SCAP-Rdy. For all $\mathcal{D}$, let $\mathcal{D}' = \mathcal{D}\{\rho \rightsquigarrow (\mathcal{L}_{\text{SCAP}}, [\![\_]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}}), \rho_r \rightsquigarrow (\mathcal{L}_{\text{RDY}}, [\![\_]\!]^{\rho_r}_{\text{RDY}})\}$. If $\psi \vdash \{(p, g)\}\,\mathtt{f}$: scheduler in SCAP-Rdy, $\Psi = \llcorner\psi\lrcorner_{\rho_r} \cup \{(\mathtt{scheduler}, (\rho, \mathcal{L}_{\text{SCAP}}, (p_s, g_s)))\}$, and $(\mathsf{sch\_pre}, \mathsf{sch\_grt}) \Rightarrow (p_s, g_s)$, we have $\mathcal{D}', \Psi \vdash \{[\![\,(p, g)\,]\!]^{\rho_r}_{\text{RDY}}\}$ call `scheduler` in OCAP-x86.

Here $[\![\_]\!]^{\rho_r}_{\text{RDY}}$ is the interpretation for SCAP-Rdy, and $(p, g) \Rightarrow (p', g')$ is defined as:

$$(\forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow p'\ \mathbb{S}) \wedge (\forall \mathbb{S}, \mathbb{S}'.\ g'\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}').$$

We do not show the details of the program logics and their interpretations. Specifications for primitives at Level C are similar to those for the AIM implementations [7]. Interested readers can refer to the Coq implementation [8] for more details.

7 Implementations in Coq

In our Coq implementations, we use a simplified version of OCAP-x86, which instantiates the full-version OCAP-x86 with a priori knowledge of the four domain-specific logics (SCAP-x86, SCAP-Rdy, SCAP-Idle and SCAP-Itr). The soundness of the framework and the embedding of the four logics have been formally proved in Coq. We have also verified most of the modules shown in Fig. 2, which have around 300 lines of x86 assembly code. The whole Coq implementation has around 82,000 lines, including 1165 definitions and 1848 lemmas and theorems. Figure 11 gives a break down of the number of lines for various components.

The implementation has taken many man-months, including the implementation of basic facilities such as lemmas and tactics for partial mappings, queues, and separation

Component	# of lines
Basic Utility Definitions & Lemmas	2,766
Machine, Opr. Semantics & Lemmas	3,269
Separation Logic, Lemmas & Tactics	6,340
OCAP-x86 & Soundness	1,711
SCAP-x86 & Soundness	1,357
Thread Queues & Lemmas	1,199
SCAP-Rdy, SCAP-Idle & SCAP-Itr	26,347

Component	# of lines
Assembly Code at Level S	292*
enQueue/deQueue	4,838
scheduler, block, unblock	7,107
Assembly Code at Level C	411*
Timer Handler	2,344
`yield` & `sleep`	7,364
locks: `acq_h` & `rel_h`	10,838
cond. var.: `wait_m` & `signal_m`	5,440

* They are the Coq source files containing the encoding of the assembly code. The real assembly code in these two files is around 300 lines.

Fig. 11. The Verified Package in Coq

logic assertions. The lesson we learn is that writing proofs is very similar to developing large-scale software systems — many software-engineering principles would be equally helpful for proof development; especially a careful design and a proper infrastructure is crucial to the success. We started to prove the main theorems without first developing a proper set of lemmas and tactics. The proofs done at the beginning used only the most primitive tactics in Coq. Auxiliary lemmas were proved on the fly and some were proved multiple times by different team members. The early stage was very painful and some of our members were so frustrated by the daunting amount of work. After one of us ported a set of lemmas and tactics for separation logic from a different project [12], we were surprised to see how the proof was expedited. The tactics manipulating separation logic assertions, especially the ones that reorder sub-terms of separating conjunctions, have greatly simplified the reasoning about memory.

Another observation is that verifying both the library (*e.g.,* Level S primitives) and its clients (*e.g.,* Level C code) is an effective way to validate specifications. We have found some of our initial specifications for Level S code are too strong or too weak to be used by Level C. Also, to avoid redoing the whole proof after fixing the specifications, it is helpful to decompose big lemmas into smaller ones with clear interfaces.

The size of our proof scripts is huge, comparing with the size of the assembly code. This is caused in part by the duplicate proof of the same lemmas by different team members. Another reason is the lack of proper design and abstraction: when an instruction is seen a second time in the code, we simply copy and past the previous proof and do some minor changes. The proof is actually developed very quickly after introducing the tactics for separation logic. For instance, the 5440 lines Coq code verifying condition variables is done by one of the authors in two days. We believe the size of proof scripts can be greatly reduced with more careful abstractions and more reuse of lemmas.

8 Related Work and Conclusions

Bevier [1] verified Kit, an OS kernel implemented in machine code, using the Boyer-Moore logic. Gargano *et al.* [10] showed a framework to construct a verified OS kernel in the Verisoft project. Both kernels support process scheduling and interrupts (and more features). Their interrupt handlers are part of the kernel, but the kernels are sequential and cannot be interrupted. There are no kernel level threads either. In both cases, it is not clear how to verify the concurrent higher-level code (processes in Kit [1] or CVMs in the Verisoft project [10]) and link it with the verified kernel as a whole system. The CLI stack project [2] verified sub-systems at different abstraction levels and composed them as a whole verified system. Interactions between different levels were formalized and verified. The verification, however, was based on operational semantics and may not be scalable to handle concurrency and higher-order code pointers. Ni *et al.* [14] verified a non-preemptive thread library in XCAP [13], which treats `pc` stored in TCBs as first-class continuations. The library, however, is not linked with verified threads. Elphinstone *et al.* [5] proposed to implement a prototype of kernel in Haskell, and to verify that the C implementation satisfies the Haskell specification.

In this paper, we present an application of our open-framework-based methodology for system verification. We verify thread implementations, synchronization primitives

and interrupt handlers using domain-specific logics and link them in the open framework to get a verified whole system. The work is based on our previous work addressing various theoretical problems, *i.e.,* the OCAP framework [6], the SCAP logic [9] and the AIM machine and logic for preemptive threads and interrupts [7]. In our previous work on OCAP [6], we showed how to link a verified scheduler with non-preemptive threads. That was a proof-of-concept example, which was not developed for real machine architecture and did not support interrupts.

There might be alternative solutions to some of the problems shown in Sect. 2. For instance, using separation logic's hypothetical frame rules [16], it is also possible to hide the concrete representations of thread queues for Level C. However, it is not clear how to support first-class code pointers and to ban certain instructions in specific scenarios in separation logic. We manage to address all these issues in a single framework. Also, our methodology is general enough to support domain-specific logics with different specification languages, *e.g.,* type systems and Hoare logics [6].

On the other hand, it is important to note that our methodology and framework do not magically solve interoperability problems for arbitrary domain-specific logics. Embedding foreign logics into the framework and letting them interact with each other involve non-trivial theoretical and engineering problems. For instance, invariants enforced in different logics need to be recognized and be properly encoded in the foundational logic. The encoding may also require mappings of program states from higher abstraction levels to lower levels. The problems may differ in specific applications. We would like to apply our methodology to larger applications (*e.g.,* verifying our 1300-line OS kernel) to further test its applicability.

Acknowledgment

We thank anonymous referees for their suggestions and comments. Wei Wang, Haibo Wang, and Xi Wang helped prove some of the lemmas in our Coq implementation. Xinyu Feng and Zhong Shao are supported in part by gift from Microsoft and NSF grant CCR-0524545. Yu Guo is supported in part by grants from National Natural Science Foundation of China (under grants No. 60673126 and No. 90718026) and Intel China Research Center. Yuan Dong is supported in part by National Natural Science Foundation of China (under grant No. 60573017), Hi-Tech Research And Development Program Of China (under grant No. 2008AA01Z102), China Scholarship Council, and Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList). Any opinions, findings, and contributions in this document are those of the authors and do not reflect the views of these agencies.

References

[1] Bevier, W.R.: Kit: A study in operating system verification. IEEE Trans. Softw. Eng. 15(11), 1382–1396 (1989)
[2] Bevier, W.R., Hunt, W.A., Moore, J.S., Young, W.D.: Special issue on system verification. Journal of Automated Reasoning 5(4), 409–530 (1989)
[3] Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. In: PLDI 2007, pp. 66–77 (June 2007)

[4] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.1
[5] Elphinstone, K., Klein, G., Derrin, P., Roscoe, T., Heiser, G.: Towards a practical, verified kernel. In: Proc. 11th Workshop on Hot Topics in Operating Systems (May 2007)
[6] Feng, X., Ni, Z., Shao, Z., Guo, Y.: An open framework for foundational proof-carrying code. In: TLDI 2007, pp. 67–78 (January 2007)
[7] Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: PLDI 2008 (to appear, June 2008)
[8] Feng, X., Shao, Z., Guo, Y., Dong, Y.: Combining domain-specific and foundational logics to verify complete software systems, extended version and Coq implementations (2008), http://flint.cs.yale.edu/flint/publications/itrimp.html
[9] Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: PLDI 2006, pp. 401–414 (June 2006)
[10] Gargano, M., Hillebrand, M.A., Leinenbach, D., Paul, W.J.: On the correctness of operating system kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603. Springer, Heidelberg (2005)
[11] Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. on Programming Languages and Systems 5(4), 596–619 (1983)
[12] McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: PLDI 2007, pp. 468–479 (June 2007)
[13] Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: Proc. 33rd ACM Symp. on Principles of Prog. Lang, pp. 320–333 (2006)
[14] Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 189–206. Springer, Heidelberg (2007)
[15] O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)
[16] O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL 2004, pp. 268–280 (January 2004)
[17] Paulin-Mohring, C.: Inductive definitions in the system Coq—rules and properties. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664. Springer, Heidelberg (1993)
[18] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. LICS 2002, pp. 55–74 (July 2002)

JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML

Patrice Chalin, Perry R. James, and George Karabotsos

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Canada
{chalin,perry,g_karab}@dsrg.org

Abstract. Tool support for the Java Modeling Language (JML) is a very pressing problem. A main issue with current tools is their architecture: the cost of keeping up with the evolution of Java is prohibitively high: e.g., Java 5 has yet to be fully supported. This paper presents JML4, our proposal for an Integrated Verification Environment (IVE) for JML that builds upon Eclipse's support for Java, enhancing it with Runtime Assertion Checking (RAC), Extended Static Checking (ESC) and Full Static Program Verification (FSPV). Though it currently only supports a subset of JML, we believe that JML*4* is the first IVE to support such a full range of verification techniques for a mainstream programming language.

1 Introduction

The Java Modeling Language (JML) is the most popular Behavioral Interface Specification Language (BISL) for Java. JML is recognized by a dozen tools and used by over two dozen institutions for teaching and/or research, mainly in the context of program verification [20]. Tools exist to support the full range of verification from Runtime Assertion Checking (RAC) to Full Static Program Verification (FSPV) with Extended Static Checking (ESC) in between [6]. In fact, JML is the only BISL supported by all three of these verification technologies.

Unfortunately, JML tools have been aging very quickly. Researchers have been unable to keep up with the rapid pace of evolution of both Java and JML. A prime example of this is the lack of support for Java 5, despite the fact that it was released in 2004. Keeping up with changes in Java is very labor intensive and from an academic researcher's point of view unrewarding.

In this paper we present JML4, an Eclipse-based Integrated development and Verification Environment (IVE) for Java and JML. Being built on top of the Eclipse Java Development Tooling (JDT), JML4 gets up-to-date support for Java almost "for free". Our contributions are as follows:

- We summarize the JML tooling state-of-affairs, reflecting upon lessons learned from the development of the first generation of tools, projecting successes into our statement of **goals** for any next generation tooling infrastructure (Section 2).

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 70–83, 2008.

- With the purpose of illustrating progress made in achieving these goals, we describe the capabilities of JML4 (Section 3), with a particular focus on recent additions (Section 3.2) which now bring to JML4 support for the full range of verification techniques (from RAC to FSPV). This makes JML4 the first IVE for a mainstream language supporting such a full range of verification techniques.
- We describe the architecture of JML4 (Section 4), allowing the reader to determine, at an architectural level, the extent to which JML4 achieves its goals.
- An assessment of the current state of JML4 is provided (Section 5). In particular we reassess the goals and identifying risk items.

The capabilities of verification tools supporting JML as well as other languages are reviewed in the section on related work (Section 6). We conclude in Section 7.

2 Problem

First Generation Tools: duplication of effort & high (collective) maintenance overhead. JML can be seen as an extension to Java that adds support for Design by Contract (DBC) though it has more advanced features—such as specification-only class attributes, support for frame properties (indicating which parts of the system state a method must leave unchanged), and behavioral subtyping—that are essential to writing complete interface specifications [12]. The main first generation JML tools essentially consist of:

- Common JML tool suite[1] also known as JML2, which includes the JML RAC compiler and JmlUnit [6],
- ESC/Java2, an extended static checker [15], and
- LOOP and PVS tool pair which supports full static program verification [25].

Of these, JML2 is the original JML tool set. Although ESC/Java2 and LOOP initially used their own annotation languages, they rapidly switched to JML.

Being independent development efforts, each of the tools mentioned above has its own front-end (e.g. scanner, parser, abstract syntax tree (AST) hierarchy and static analysis code) essentially for *all* of Java and JML. This amounts to substantial duplication of effort and code. Recent evolution in the definition of Java (e.g. Java 5, especially generics) and of JML made it painfully evident that the limited resources of the JML community could not cope with the workload that it engendered.

As a result, for example, none of the current first generation tools can yet fully support Java 5 features. With respect to the evolution of JML, only JML2 supports the new non-null by default semantics [9].

Moving Forward with Lessons Learned. What lessons can be learned from the development of the first generation of tools, especially JML2, which (since early 2000) has been the reference implementation of JML? JML2 was essentially developed as an extension to the MultiJava (MJ) compiler. By "extension", we mean that: for the most part, MJ remains independent of JML; many JML features are naturally implemented

[1] Formerly the Iowa State University (ISU) JML tool suite.

by subclassing MJ features and overriding methods; in other situations, extension points (calls to methods with empty bodies) were added to MJ classes so that it was possible to override behavior in JML2. We believe that this approach has allowed JML2 to be successfully maintained as the JML reference implementation until recently. Then what went wrong? We believe it was a combination of factors including the advent of a relatively big step in the evolution of Java (including Java 5 generics) and the difficulty in finding developers to upgrade MJ.

Goals for Next Generation Tool Bases. Keeping in mind that we are targeting mainstream industrial software developers as our primary user base, our goals for a next generation research vehicle for the JML community can be summarized as follows: the new tooling infrastructure should be [22]:

1) Based on, at least a Java compiler, ideally a modern IDE, whose maintenance is *outside* the JML community;
2) Built, to the extent practicable, as a decoupled "extension" of the base so as to *minimize the integration* effort required when new versions of the base compiler/IDE are released;
3) Capable of supporting *at least* the integrated capabilities of RAC, ESC, and FSPV

As will be discussed in the section on related work, a few recent JML projects have attempted to satisfy these goals. In the sections that follow, we describe how we have attempted to satisfy them in our design of JML4.

3 JML4

3.1 Early Prototype

After much discussion, both within our own research group and with other members of the JML community, we decided that basing a next generation JML tooling framework on the Eclipse JDT seemed like the most promising approach. While the JDT is large—approximately 1 MLOC for 5000 files—and the learning curve is steep (partly due to lack of documentation) we nonetheless chose to take the plunge and began prototyping JML4 in 2006.

In our first feature set, JML4 enhanced Eclipse 3.3 with: scanning and parsing of nullity modifiers (`nullable` and `non_null`), enforcement of JML's non-null type system (both statically and at runtime) [9] and the ability to read and make use of the extensive JML API library specifications. This architecturally significant subset of features was chosen so as to exercise some of the basic capabilities that any JML extension to Eclipse would need to support. These include

- Recognizing and processing JML syntax inside specially marked comments, both in `*.java` files as well as `*.jml` files;
- Storing JML-specific nodes in an extended AST hierarchy,
- Statically enforcing a modified type system, and
- Generating runtime assertion checking (RAC) code.

The chosen subset of features was also seen as useful in its own right [9], somewhat independent of other JML features. In particular, the capabilities formed a natural extension to the existing embryonic Eclipse support for nullity analysis.

This early prototype served as a basis for analysis by members of the JML Reloaded "subcommittee" of the JML Consortium. In conclusion, the decision was to move forward with development of JML4 [22].

3.2 New Feature Set

In this section we describe the current JML4 feature set as evidence that progress is being made towards the goals stated in Section 2, especially with respect to framework capabilities in support of the full range of verification technologies.

3.2.1 Overview

We mention in passing that in parallel with our work on next generation components we have integrated the two main first-generation JML tools: ESC/Java2 and the JML RAC. Hence, at a minimum, JML users actively developing with first generation tools will be able to continue to do so, but now within the more hospitable environment offered by Eclipse.

With respect to the next generation components proper, at the time of writing, JML4's front-end support is nearing what is called JML Level 1, which includes the most frequently used core JML constructs [21, §2.9]. When completed, this will provide the capabilities of a type checker, similar to that provided by JML2's `jmlc`. While basic support for RAC (e.g., inline assertions and simple contracts) is available, a next generation design inspired from the current JML compiler is being lead by its original author [13], Yoonsik Cheon, and his team at the University of Texas at El Paso. Our research group is leading development in static verification components—details are given in the next section. Yet others are exploring the integration of new tools for JML; e.g. Robby and his team at Kansas State University are looking into the integration of the Bogor/Kiasan symbolic execution system and the associated KUnit test generation framework [16]. We are hopeful that next generation components fully processing JML Level 1 specifications will be ready by the end of 2008.

3.2.2 Static Verification (SV): Ground-Up Designs Using Latest Techniques

Besides work on the JML4 infrastructure, our research group has been focusing its efforts on the development of a new component called the JML Static Verifier (JML SV). This new component offers the basic capabilities of ESC and FSPV.

The ESC component of JML4, referred to as ESC4, is a ground-up rewrite of ESC which is based on Barnett and Leino's innovative and improved approach to a weakest precondition semantics for ESC [4]. Our FSPV tool, called the FSPV Theory Generator (TG), is like the JML LOOP compiler [25] in that it generates theories containing lemmas whose proof establish the correctness of the compilation unit in question. The FSPV TG currently generates theories written in the Hoare Logic of SIMPL—an Isabelle/HOL based theory designed for the verification of sequential imperative programs [23]. Lemmas are expressed as Hoare triples. To prove the correctness of such lemmas, a user can interactively explore their proof using the Eclipse version of Proof General (PG) [2]—see Figure 1.

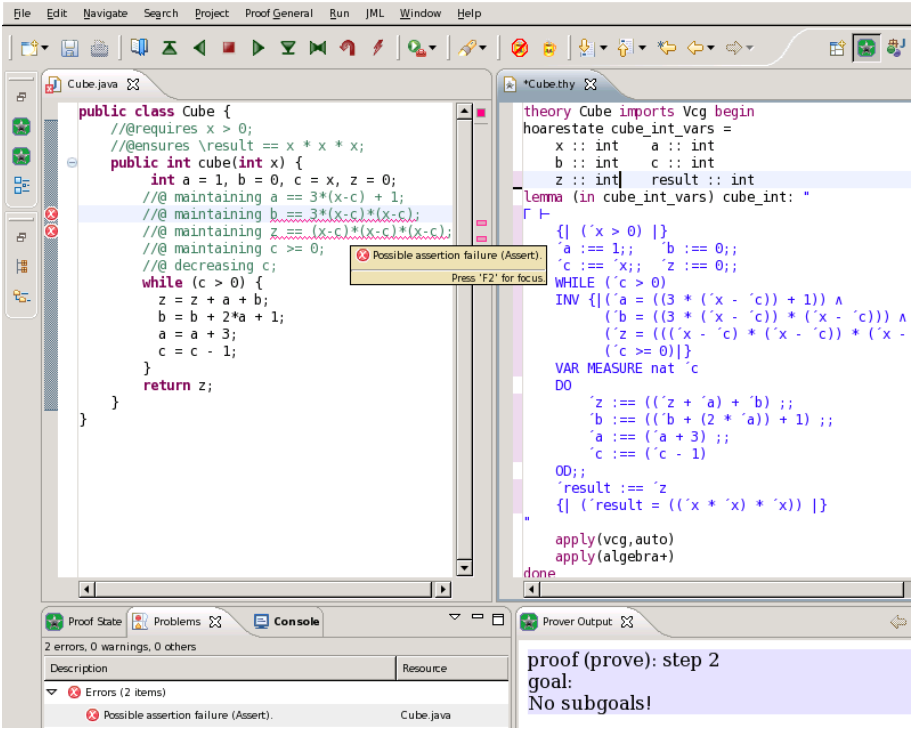


Fig. 1. ESC4 reporting that it cannot prove loop invariants in Cube.java (because it is using only first order provers). FSPV TG theory (Cube.thy) and its proof confirmed valid by Isabelle.

3.2.3 Innovative SV Features

In addition to supporting ESC and FSPV, the JML SV currently supports the following features, most of which are novel either in the context of verification tools in general or JML tools in particular:

- Multi Automated Theorem Prover (ATP) support including:
  - First-order ATPs: Simplify and CVC3.
  - Isabelle/HOL, which, we have found can be used quite effectively as an ATP.
- A technique we call 2D Verification Condition (VC) cascading where VCs that are unprovable are broken down into sub-VCs (giving us one axis of this 2D technique) with proofs attempted for each sub-VC using each of the supported ATPs (second axis).
- VC proof status caching. VCs (and sub-VCs) are self-contained, context-independent lemmas (because the lemmas' hypotheses embed their context), and hence they are ideal candidates for proof status caching. I.e., the JML SV keeps track of proven VCs and reuses the proof status on subsequent passes, matching textually VCs and hence avoiding expensive re-verification.
- Offline User Assisted (OUA) ESC, which we explain next.

By definition, ESC is fully automatic [18] whereas FSPV requires interaction with the developer. OUA ESC offers a compromise: a user is given an opportunity to provide (offline) proofs of sub-VCs which ESC4 is unable to prove automatically. Currently, ESC4 writes unprovable lemmas to an Isabelle/HOL theory file (one per compilation unit). The user can then interactively prove the lemmas using Proof General. Once this is done, ESC4 will make use of the proof script on subsequent invocations. We have found OUA ESC to be quite useful because ESC4 is generally able to automatically prove most sub-VCs, hence only asking the user to prove the ones beyond ATP abilities greatly reduces the proof burden on users.

Figure 2 sketches the relationship between the effort required to make use of each of the JML SV verification techniques vs. the level of completeness that can be achieved. Notice how ESC4, while requiring no more effort to use than its predecessor ESC/Java2, is able to achieve a higher level of completeness. This is because ESC4 makes use of multiple prover back-ends including the first order provers Simplify and CVC3 as well as Isabelle/HOL. As was mentioned earlier, Isabelle/HOL can be used quite effectively as an automated theorem prover; in fact, Isabelle is able to (automatically) prove the validity of assertions that are beyond the capabilities of the first order provers—e.g., assertions making use of quantifiers. An example of a method which JML SV can prove correct using Isabelle/HOL as an ATP is Cube.java given in Figure 1 (the reason ESC4 shows that it is unable to prove the loop invariants is because we disabled use of Isabelle/HOL as an ATP for illustrative purposes—to contrast with what can be proven using Cube.thy).

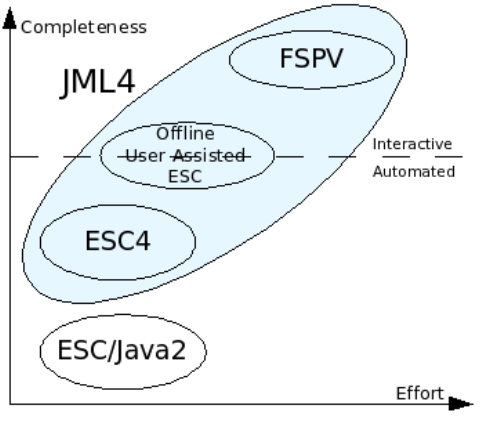


Fig. 2. Static verification in JML4

In summary, the latest features added to JML4 make it the first IVE for a mainstream programming language to support the full range of verification technologies (from RAC to FSPV). Its innovative features make it easier to achieve complete verification of JML annotated Java code and this more quickly: preliminary results show that ESC4 will be at least 5 times faster than ESC/Java2. Furthermore, features like proof caching, and other forms of VC proof optimization, offer a further 50% decrease in verification time. Of course, until JML SV supports the full JML language, these results are to be taken as preliminary, but we believe that they are indicative of the kinds of efficiency improvements that can be expected.

In the next section, we explore the architecture of JML4 in general, and the JML SV in particular, allowing us to see how the features described above have been realized.

4 Architecture

In this section we present an architectural overview of JML4 with a particular focus on the compiler (rather than other aspects of the IDE) which is referred to as the JML4 *core*.

4.1 Overview

At the heart of JML4 is the JML4 core, whose processing phases are illustrated in Figure 3. Most phases are conventional. In the Eclipse JDT (and hence in JML4), there are two types of parsing: in addition to the usual full parse, there is also a diet parse, which only gathers class signature information and ignores method bodies. JML4-specific phases are shown in bold and include the merge of external specifications and static verification. Code instru-mentation for the purpose of Runtime Assertion Checking (RAC) is done during the JDT's code generation phase.

A top-level module view of Eclipse and JML4 is given in Figure 4. Eclipse is a plug-in based application platform and hence an Eclipse application consists of the Eclipse plug-in loader (Platform Runtime component), certain common plug-ins (such as those in the Eclipse Platform package) along with application specific plug-ins. While Eclipse is written in Java, it does not have built-in support for Java. Like all other Eclipse features, Java support is provided by a collection of plug-ins referred to as the Eclipse Java Development Tooling (JDT).

The JML JDT extends the Eclipse JDT to offer basic support for JML. In particular, the JML JDT contains a modified scanner, parser and Abstract Syntax Tree (AST) hierarchy. The JML Static Verifier (SV) component design is described next.

4.2 JML Static Verifier (SV)

As was explained in the previous section, the JML SV supports two

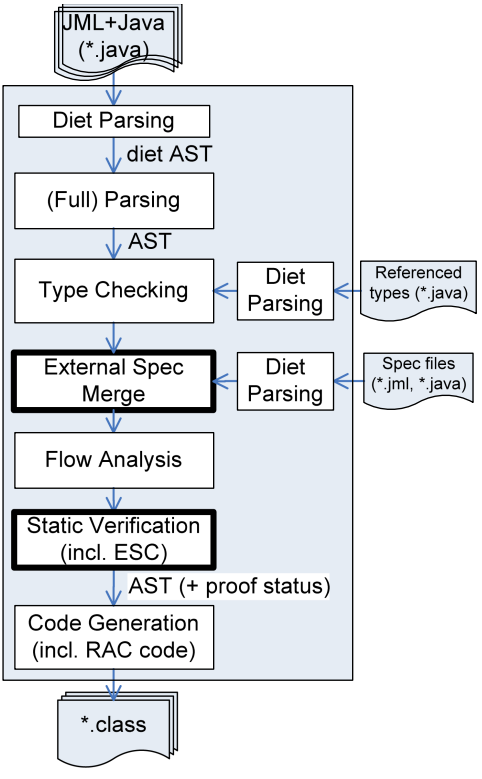


Fig. 3. JDT/JML4 core phases (phases only present in JML4 are in bold)

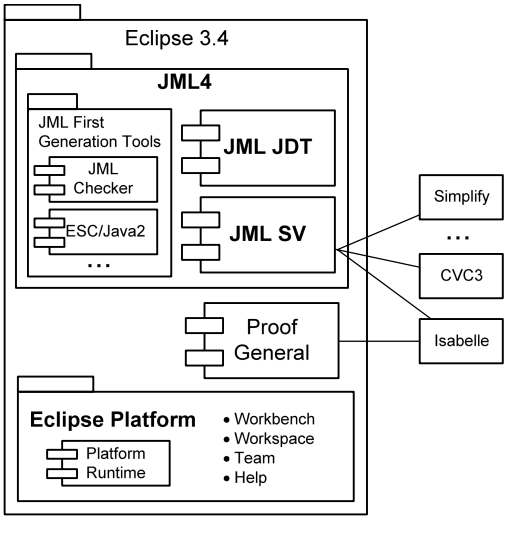


Fig. 4. Eclipse and JML4 component diagram

main kinds of verification: extended static checking—both the normal kind and Offline User Assisted (OUA) ESC—and full static program verification. These are realized by the subcomponents named ESC4 and the FSPV Theory Generator (TG), respectively. A diagram illustrating dataflow for the JML SV is given in Figure 5. The input to the JML SV is a fully resolved AST for the compilation unit actively being processed.

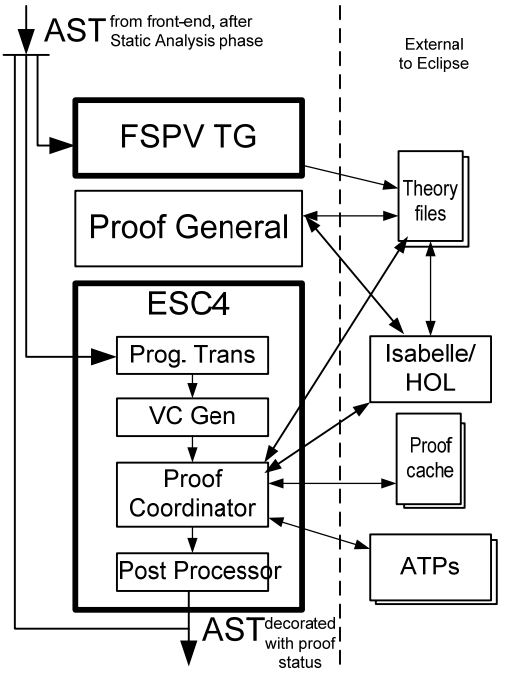


Fig. 5. JML SV dataflow

Initiated on user request, separate from the normal compilation process, the FSPV TG generates Isabelle/HOL theory files for the given compilation unit (CU). One theory file is generated per CU. The theory files can then be manipulated by the user using Proof General.

When activated (via compiler preferences), ESC4 functionality kicks in during the normal compilation process following standard static analysis. The ESC phases are the standard ones [18], though the *approach* used by ESC4 is new in the context of JML tooling: it is a realization of the Barnett and Leino approach [4] used in Spec# in which the input AST is translated into VCs by using a novel form of guarded-command program as an intermediate representation. The Proof Coordinator decides on the strategy to use: e.g. single prover, cascaded VC proofs or OUA ESC. In the latter case, Isabelle theory files are consulted when sub-VCs are unprovable and a user-supplied proof exists. Unfortunately, the detailed design and full details of the behavior of the JML SV are beyond the scope of this paper.

5 Assessment

Goals. In Section 2, we listed three goals to be satisfied by any next generation tooling infrastructure for JML. In summary, (1) the infrastructure should be built as an extension to a compiler + IDE whose maintenance is guaranteed by others, (2) minimize integration efforts as the IDE code base evolves, and (3) demonstrate the feasibility of supporting the full range of current verification technologies.

The first goal has been achieved simply by our choice of the Eclipse JDT as a tool base—though doing so introduced some risks which we discuss in the next section. Since the second goal (ease of maintenance) is related to the risk items, we defer assessment of this goal to the next section as well. By the implementation of the most

recent JML4 feature set we have achieved, for a non-trivial subset of JML, support for RAC, ESC and FSPV, thus demonstrating the third goal.

As evidence that JML4 is actually usable in practice, we point out that we have successfully applied it to a case study (totaling over 470K SLOC) in which we made use of the enhanced non-null-type static analysis and RAC capabilities of JML4 [9]. Furthermore, we are also currently capable of compiling first generation tools (e.g. ESC/Java2) using the first generation tools themselves within JML4.

Risk items. The Eclipse JDT is a very dynamic code base, with code changes introduced every few weeks. This was perceived as an important risk item for JML4. As we explain next, we believe that we have found a suitable means of allowing JML4 to extend the JDT so as to keep JML4 code rework to a minimum when JDT changes are released.

JML4, like JML2, is built as a closely integrated and yet (relatively) loosely coupled extension to an existing compiler. An additional benefit for JML4 is that the timely compiler-base maintenance is assured by the Eclipse Foundation developers. Hence, as compared to JML2, we have traded in committer rights for free maintenance; a choice which we believe will be more advantageous in the long run—in particular due to the rapid pace of the evolution of Java. Unfortunately, loosing committer rights means that we must maintain our own mirror of the JDT code.

While we originally had the goal of creating JML4 as a proper Eclipse plug-in, only making use of public JDT APIs (rather than as a replacement plug-in for the JDT), it rapidly became clear that this would result in far too much copy-and-change code; so much so that the advantage of coupling to an existing compiler was lost (e.g. due to the need to maintain our own full parser and AST). Nonetheless we were also originally reluctant to build atop internal APIs, which contrary to public APIs, are subject to change—with weekly releases of the JDT code, it seemed like we would be building on quicksand. Anticipating this, we established several conventions that make merging in the frequent JDT changes both easier and less error prone. These include

- Avoiding introducing JML features by the copy-and-change of JDT code, instead we make use of subclassing and method extension points;
- Bracketing any changes in our copy of the JDT code with special comment markers; and
- Segregating all JML-specific fields and methods at the end of the file.

While following these conventions, incorporating each of the regular JDT updates since 2006 has taken less than 15 minutes, on average. While we believe that further decoupling from the JDT is possible, we feel it is too early to undertake this since we have yet to experience the integration of the full JML feature set.

One of our objectives is also to make it easy for all members of the JML research community to extend JML4, e.g., to integrate their own tools. JML developers who have been initiated to JML4 development have expressed concern in this respect due to the complexities of the JDT which developers are currently exposed to. With more experience in the development of JML4, we are hopeful that it will be possible to hide some of the unnecessary JDT complexities.

Tool Scope and moving forward. Though the JML4 infrastructure and the JML SV are at a preliminary stage of development they support, we believe, a rudimentary yet useful

subset of JML. E.g., verification is implemented for the normal behavior (no exceptional behavior) of simple method contracts, without behavioral subtyping. Though we cannot use JML SV on the full ESC/Java2 source, ESC/Java2 can be compiled with JML4's static and runtime processing of nullity constraints. This in itself has proven useful in indentifying bugs with ESC/Java2 code and annotations [11].

Reengineering the first generation tool base while creating new tools and elaborating the tooling infrastructure is going to take time. Thankfully, with the increased contributions by other JML research teams, we are hopeful that feature increments will be added on a regular basis so that by year's end, JML4 will have achieved a level of support for JML that exceeds the capabilities of the first generation tools.

6 Related Work

6.1 Verification Tool Support for Java and/or JML

In this section we briefly compare JML4 to its sibling next generation projects JML3, JML5, JaJML as well as to the Java Applet Correctness Kit (JACK). Further details, examples and tools are covered in [10].

JML3. The first next-generation Eclipse-based initiative was JML3, created by David Cok. The main objective of the project was to create a proper Eclipse plug-in, independent of the internals of the JDT [14]. Considerable work has been done to develop the necessary infrastructure, but there are growing concerns about the long term costs of this approach.

Because the JDT's parser is not extensible from public JDT extensions points, a separate parser for the entire Java language and an AST had to be created for JML3; in addition, Cok notes that "JML3 [will need] to have its own name / type / resolver / checker for both JML constructs [and] all of Java" [14]. Since one of the main goals of the next generation tools is to escape from providing support for the Java language, this is a key disadvantage.

JACK. The Java Applet Correctness Kit (JACK) is a tool for JML annotated Java Card programs initially developed at Gemplus (2002) and then taken over by INRIA (2003) [5]. It uses a weakest precondition calculus to generate proof obligations that are discharged automatically or interactively using various theorem provers [8]. While JACK is emerging as a candidate next generation tool (offering features unique to JML tools such as verification of annotated byte code [7]), being a proper Eclipse plug-in, it suffers from the same drawbacks as JML3 with respect to the need to maintain a separate compiler front-end. Additionally JACK does not provide support for RAC which we believe is an essential component of a mainstream IVE. An advantage that JACK has over JML4's current capabilities is that it can present VCs to the user in a Java/JML-like notation.

JML5. The JML5 project has taken a different approach [24]. Its goal is to embed JML specifications in Java 5 annotations rather than Java comments. Such a change will allow JML's tools to use any Java 5 compliant compiler. Unfortunately, the use of annotations has important drawbacks as well. In addition to requiring a separate

Table 1. A Comparison of Possible Next Generation JML Tools

		JML2	JML3	JML4	JML5	JaJML	EJ2	JACK
Base Compiler / IDE	**Name**	MJ	JDT	JDT	any Java 7+	JastAdd Java	EJ2	JDT
	Maintained (supports Java ≥ 5)	×	✓	✓	✓	✓	×[1]	✓
Reuse/extension of base (e.g. parser, AST) vs. copy-and-change		✓	×	✓	×	✓	×	×
Tool Support	**RAC**	✓	✓	✓	(✓)	✓	N/A	N/A
	ESC	N/A	(✓)	✓	N/A	×	✓	✓[2]
	FSPV	N/A	×	✓	N/A	×	N/A	✓

MJ = MultiJava, JDT = Eclipse Java Develoment Toolkit EJ2 = ESC/Java2
N/A = not possible, practical or not a goal, (✓) = planned

[1] ESC/Java2 is being maintained, but its compiler front end has yet to reach Java 5.

[2] Strictly speaking, JACK supports an automated form of FSPV, not ESC.

parser to process the JML specific annotation contents (e.g. assertion expressions), Java's current annotation facility does not allow for annotations to be placed at all locations in the code at which JML can be placed. JSR-308 is addressing this problem as a consequence of its mandate, but any changes proposed would only be present in Java 7 and would not allow support for earlier versions of Java [17].

JaJML. JaJML is an early research prototype built atop JastAdd Java compiler [19]. JastAdd is a compiler framework that uses attribute grammars and supports Aspect Oriented Programming. JaJML's main advantage over JML4 is the ease with which it can be extended. Indeed, Haddad and Leavens note that adding RAC support to JaJML for while loop variants and invariants was done in a fraction of the number of lines of code that were needed to add them to JML4. The main disadvantages of JaJML include its lack of integration with an IDE and no guaranteed third-party maintenance for the underlying JastAdd Java compiler. While use of JastAdd offers increased modularity, there is likely to be a performance impact. The ability of JaJML to provide support for static verification has yet to be derisked.

Table 1 presents a summary of the comparison of the tools supporting JML. As compared to the approach taken in JML4, the main drawback of the other tools is that they are likely to require more effort to maintain over the long haul as Java continues to evolve and due to the looser coupling with their base.

6.2 Verification Tool Support for Other Languages

KeY. While the KeY tool was recently adapted to accept JML, it also supports other languages [1]. KeY is an integrated development environment which targets verification at a slightly higher level than most other tools. This is because KeY supports the annotation of design artifacts such as class diagrams. KeY does not support RAC or ESC though both automated and interactive FSPV are supported. Like JACK, KeY presents VCs in a JML-like notation.

OmnibusOmnibus. Omnibus is a functional language with syntax similar to Java's that compiles to JVM bytecode [26]. The language was designed with reduced capabilities as compared to Java (e.g. it lacks support for exceptions, interface inheritance, and concurrency) so as to ease verification. Each of the source files in an Omnibus project has an associated verification policy that gives the level of verification required for it, which can be RAC, ESC, or FSPV. A custom IDE was developed that make these and other development activities easier, including tracking the verification status of (and method used for) each file. Simplify and PVS are the theorem provers used for ESC and FPV, respectively. Hence, Omnibus offers verification support comparable to that of JML4, though not for a mainstream language.

SPARK. SPARK [3] was developed for the implementation of safety-critical control systems. It is a subset of Ada extended with annotations to provide support for (an enriched form of) DBC. The subset was chosen to be amenable to ESC and FSPV and yet be useful for writing industrial applications. Unlike the other languages discussed in this section, there is no support for RAC, since static verification is used to show that errors—including contract violations—cannot happen at runtime. Static analysis is performed in three stages. The first stage is provided by the Examiner and is similar to the JDT's flow analysis. In the second stage, the Simplifier automatically discharges those VCs that it can and leaves the rest for the Proof Checker, an interactive theorem prover. The Proof Obligation Summary (POGS) tool is used to reduce the various outputs of the static analysis and proof tools to a single report that gives the status of the verification process overall including, in particular, a list of any VCs that remain unproved. In a sense, when using Offline User Assisted ESC, JML4 can be seen to behave like the POGS. All of these tools are stand-alone command-line tools.

7 Conclusion and Future Work

The idea of providing JML tool support by means of a closely integrated and loosely coupled extension to an existing compiler was successfully realized in JML2. Although this worked well, unfortunately the chosen base Java compiler is no longer officially maintained. Applying the same approach we have extended the Eclipse JDT to create the base infrastructure of JML4.

The first JML4 prototype served as a basis for discussion by some members of the JML consortium, and eventually it came to be adopted as the main avenue to pursue in the JML Reloaded effort [22]. A JML Winter School followed in February of this year, during which members of the community were given JML4 developer training [20, Wiki]. Since then, we have enhanced JML4's feature set, in particular, with support for next generation ESC and FSPV components.

Even though JML4's approach is currently more invasive than a proper plug-in design, using this approach we have since 2006, been able to (i) maintain JML4 despite the continuous development increments of the Eclipse JDT, and (ii) demonstrated, through the recent addition of the JML SV, that JML4's infrastructure is capable of supporting the full range of verification approaches from RAC to FSPV. Hence, we are hopeful, that JML4 will be a strong candidate to act as a next generation research platform and industrial grade verification environment for Java and JML.

Of course, there is much work yet to be done. The most pressing is completing the compiler front-end to support all or most of JML. Once this is done, it will be necessary to propagate support for JML constructs into the RAC, ESC and FSPV components. Completion of the front-end will also mark a milestone after which developers of other JML tools will be able to explore the possibility of integrating their tools within the JML4 framework.

Acknowledgments

We thank the anonymous referees for their comments and insightful questions. We also thank Gary Leavens for hosting the first JML Winter School which acted as the kick-off developer meeting for JML4, as well as Jooyong Lee and Daniel Zimmerman for their continued contributions to JML4. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Québec Fonds de Recherche sur la Nature et les Technologies.

References

[1] Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY Tool. SoSyM 4, 32–54 (2005)

[2] Aspinall, D.: Proof General (2008), `http://proofgeneral.inf.ed.ac.uk`

[3] Barnes, J.: High Integrity Software: The Spark Approach to Safety and Security. AW (2003)

[4] Barnett, M., Leino, K.R.M.: Weakest-Precondition of Unstructured Programs. In: Workshop on Program Analysis for Software Tools and Engineering (PASTE), Lisbon, Portugal. ACM Press, New York (2005)

[5] Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., Requet, A.: JACK: a tool for validation of security and behaviour of Java applications. In: Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO) (2007)

[6] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An Overview of JML Tools and Applications. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 212–232 (2005)

[7] Burdy, L., Huisman, M., Pavlova, M.: Preliminary Design of BML: A Behavioral Interface Specification Language For Java Bytecode. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 215–229. Springer, Heidelberg (2007)

[8] Burdy, L., Requet, A., Lanet, J.-L.: Java Applet Correctness: A Developer-Oriented Approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 422–439. Springer, Heidelberg (2003)

[9] Chalin, P., James, P.R.: Non-null References by Default in Java: Alleviating the Nullity Annotation Burden. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609. Springer, Heidelberg (2007)

[10] Chalin, P., James, P.R., Karabotsos, G.: An Integrated Verification Environment for JML: Architecture and Early Results. In: Proceedings of the Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS), Cavtat, Croatia, September 3-4, pp. 47–53. ACM, New York (2007)

[11] Chalin, P., James, P.R., Rioux, F., Karabotsos, G.: Towards a Verified Software Repository Candidate: Cross-Verifying a Verifier, Concordia University, Dependable Software Research Group Technical Report (2008)
[12] Chalin, P., Kiniry, J., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
[13] Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language, Iowa State University, Ph.D. Thesis, also TR #03-09 (April 2003)
[14] Cok, D.R.: Design Notes (Eclipse.txt) (2007), `http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/trunk/docs/eclipse.txt`
[15] Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
[16] Deng, X., Robby, Hatcliff, J.: Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems, Kansas State University (2007)
[17] Ernst, M., Coward, D.: Annotations on Java Types, JCP.org., JSR 308, October 17 (2006)
[18] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), vol. 37(5), pp. 234–245. ACM Press, New York (2002)
[19] Haddad, G., Leavens, G.T.: Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations, University of Central Florida CS-TR-08-05 (April 2008)
[20] Leavens, G.T.: The Java Modeling Language (JML) (2007), `http://www.jmlspecs.org`
[21] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual (2007), `http://www.jmlspecs.org`
[22] Robby, P.C., Cok, D.R., Leavens, G.T.: An Evaluation of The Eclipse Java Development Tools (JDT) as a Foundational Basis for JML Reloaded (2008), `http://jmlspecs.svn/reloaded/planning`
[23] Schirmer, N.: A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment. In: Isabelle Archive of Formal Proofs (2008)
[24] Taylor, K.B.: A specification language design for the Java Modeling Language (JML) using Java 5 annotations. Masters thesis, Iowa State University (2008)
[25] van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
[26] Wilson, T., Maharaj, S., Clark, R.G.: Omnibus: A Clean Language and Supporting Tool for Integrating Different Assertion-Based Verification Techniques. In: Proceedings of REFT 2005, Newcastle, UK (July 2005)

Incremental Benchmarks for Software Verification Tools and Techniques

Bruce W. Weide[1], Murali Sitaraman[2], Heather K. Harton[2], Bruce Adcock[1], Paolo Bucci[1], Derek Bronish[1], Wayne D. Heym[1], Jason Kirschenbaum[1], and David Frazier[2]

[1] The Ohio State University, Columbus, OH 43210, USA
{weide,adcockb,bucci,bronish,heym,kirschen}@cse.ohio-state.edu
[2] Clemson University, Clemson, SC 29634, USA
{murali,hkeown,dfrazie}@cs.clemson.edu

Abstract. This paper proposes an initial catalog of easy-to-state, relatively simple, and incrementally more and more challenging benchmark problems for the Verified Software Initiative. These benchmarks support assessment of verification tools and techniques to prove total correctness of functionality of sequential object-based and object-oriented software. The problems are designed to help evaluate the state-of-the-art and the pace of progress toward verified software in the near term, and in this sense, they are just the beginning. They will allow researchers to illustrate and explain how proposed tools and techniques deal with known pitfalls and well-understood issues, as well as how they can be used to discover and attack new ones. Unlike currently available benchmarks based on "real-world" software systems, the proposed challenge problems are expected to be amenable to "push-button" verification that leverages current technology.

1 Introduction

> This, however, is my doctrine: whoever would learn to fly must first learn to stand and walk and run and climb and dance: one cannot fly into flying!
>
> — Friedrich Nietzsche, in *Thus Spake Zarathustra*, Third Part, 1884

> You can't get to the moon by climbing a tree.
>
> — William F. Ogden, paraphrase of S.E. Dreyfus in *Mind Over Machine*, p. 10, 1986

As the two views above reveal, there are (at least) two kinds of ambitions: ones for which incremental gains gradually lead to their achievement, and ones for which no merely incremental gains can possibly lead to their achievement.

Is it possible to reach incrementally the ultimate goal of the Verified Software Initiative (VSI), i.e., routine verification of practical software systems? There are certainly paths that are doomed not to reach this target no matter how much incremental progress can be made. For example, a proof system that is inherently not modular simply is not going to scale up to verification of practical software systems of the sort

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 84–98, 2008.

the VSI needs to tackle—regardless of how much incremental progress is made on its details. Yet history suggests that the lofty goals of the VSI, like similar goals in other disciplines, are more likely to be achieved by repeated acts of "standing on the shoulders of giants" (and others) than by an ambitious tree-climber suddenly striking out in a space ship and flying to the moon.

The benchmarks proposed in this paper, therefore, define strategically placed milestones along a path toward the goal of verified software. The spacing and nature of benchmarks to define these milestones have been *designed* to capture the anticipated incremental nature of the community effort. Two criteria have guided their development:

- They should be similar enough that incremental progress toward the goal can be recognized—and explained to the community as simply as possible.
- They should be designed especially for verification, so that progress in overcoming specific known barriers to achieving the goal can be readily demonstrated—and explained to the community as simply as possible.

1.1 Contributions

The primary contribution of this paper is its proposed set of eight *purposefully designed, incremental, early benchmarks for tools and techniques to prove total correctness of sequential object-based and object-oriented software*. These benchmarks come with a classification of the issues they raise for automated software verification and a methodology for reporting progress. They are intended to precede (in difficulty) and complement rather than replace earlier challenge problems (e.g., [1,2,3,4]); they certainly should not be seen as diminishing the significance of claimed solutions to those problems (e.g., [3,5,6]). The proposed benchmarks differ from earlier challenge problems in key respects. Most important, they are far simpler. If a system cannot be used to verify the proposed benchmarks, it is unlikely to be powerful enough to verify software that operates an e-commerce site, or implements a file system, or controls a non-trivial embedded device such as a heart pacemaker. Put otherwise, if a verification system is capable of solving complex real-world benchmarks, then there should be no problem applying it to these simpler benchmarks and using this opportunity to explain how the approach overcomes each of a number of known bumps in the road for software verification.

The proposed benchmarks are also highly focused, so tools and techniques that are not intended to address every aspect of a real-world software system can still be applied and evaluated for some or all of these benchmarks. This means that individuals or groups who are able to contribute a few pieces of the big puzzle are empowered to play. Solutions to these benchmarks can become verified software in the VSI repository. Once several systems using different languages, approaches, etc., have been applied to a given benchmark, it will be easier for the community to evaluate their relative merits and eventually to design new tools and techniques that leverage the best attributes of different efforts.

The above features also mean it is reasonable to hope for complete automation and "push-button" technology in benchmark solutions. Once humans formalize the requirements in a benchmark problem statement into specifications (supported as necessary by new or extended mathematical theories), write code (annotated as necessary

with assertions expected by the proposed technique), and provide any other necessary inputs, no further human intervention should be needed to complete the verification. The result might be a simple "proved" or "not proved", or in the latter case it might be reported in a form that simplifies debugging.

1.2 Limitations

There certainly are other benchmarks that would be appropriate for the territory covered by the ones proposed here. We do not attempt to justify that these are the "best" ones, only that they are a decent starting point. We do not even begin to suggest benchmarks that would help assess progress on other paths leading toward the VSI goal, or later benchmarks on this path: proving properties entailed by, but other than, total functionality correctness (e.g., absence of null dereferences); performance correctness (e.g., compliance with execution time or memory usage specifications); concurrency issues; distributed systems issues; graphical user interface issues; meta-level issues such as soundness and relative completeness of proof systems; proofs in programming language meta-theory [7]; and so on. The proposed new benchmarks venture nowhere near the latter areas.

There is not necessarily a total order among the issues that are raised by the proposed benchmarks. One might be able to solve a later benchmark problem before an earlier one. The order shown is consistent with a natural partial order among the primary issues that we identify as important problems to be addressed by the proposed benchmarks. In fact, some of these benchmarks have already been "solved", though we do not cite such claims. This is fine; the more different solutions the better, so the community can compare their pros and cons. And if someone chooses to start from scratch rather than building on other work to attack some of these challenges, then there is relatively little risk of massive lost effort, but there might be potentially high payoff: all the proposed benchmarks are close to the beginning of the journey and might help illustrate new ideas that are not derived as variations on previous work.

The rest of the paper is organized as follows. Section 2 presents the benchmark problem requirements statements, identifies the major issues related to each, and lists some variations of each challenge that might be considered by those proposing solutions. Section 3 describes two possible solutions to one part of the simplest benchmark, so prospective benchmark solvers have a couple of examples of what we mean. Section 4 reviews and concludes.

2 Benchmark Problems

All the proposed benchmark problems require that tools and techniques purported to solve them should have the following features:

- The proposed solution should include:
    - all formal specifications relevant to the benchmark problem requirements, including mathematical definitions, theories, and similar artifacts developed for and/or used in the specifications;
    - all code subjected to the verification process;

  - all verification conditions (VCs) involved in the verification process;
  - descriptions of the verification system proof rules employed, tools used, and techniques applied.

  As some of this information may be unwieldy or impossible to include in an exposition of acceptable length, a web site with details should be provided so readers and reviewers can check any details they deem important. (Section 3 contains links to such sites for the example solutions outlined there.)
- The proposed solution should involve both an automatic proof of total correctness of a correct solution, and evidence that the tools and techniques can automatically detect that a "slightly" incorrect solution is incorrect. Specifically, in addition to verified correct code, a benchmark solution should present a perturbed version of the code—which plausibly could have been written to meet the same specification—along with a demonstration that a bug is found automatically. For example, perhaps the proposed solution generates meaningful counterexamples as test cases that lead to failure for defective code. This requirement also could be regarded as a kind of mutation test.
- The proposed verification approach should be modular. In other words, a proof that a program unit *P* implements a specification *S* should be based only on *P*, *S*, and the specifications of the program units that *P* depends on (not their implementations). Moreover, the verification should need to be performed only once, not again for every context in which *P* is used for *S*.
- Verified software from benchmark solutions should be submitted formally to the VSI repository [8].

2.1 Benchmark #1: Adding and Multiplying Numbers

Problem Requirements: Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition, using the first operation to do the addition. Make one algorithm iterative, the other recursive.

Issues: Addition is straightforward. It is a single operation and involves a single numeric type: either integers or natural numbers, which may be a built-in type in the programming language or a user-defined abstract data type (ADT). It also involves simple specifications, and presumably results in VCs from a decidable mathematical theory (Presburger arithmetic). Multiplication explicitly adds the requirement for modularity for a program unit that is a single operation, i.e., procedural or functional abstraction, and enough richness that the underlying mathematics is undecidable. Since one operation involves iteration and the other recursion, loop invariants and termination both become concerns. There are alternative algorithms for multiplication based on addition, e.g., the Egyptian algorithm [9] as well as the more obvious and familiar ones. All the standard procedural programming constructs are likely to arise in these two pieces of code.

Variations: Other operations on integers or natural numbers, e.g., computing powers and roots, involve similar issues and should illustrate similar verification capabilities. The programming type used for numbers may or may not be bounded. Numerical

problems of this kind that involve floating point numbers are very interesting as well, raising especially difficult issues about how to specify their behavior that do not arise with either bounded or unbounded integers or natural numbers.

2.2 Benchmark #2: Binary Search in an Array

Problem Requirements: Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.

Issues: Arrays are the traditional first "collection" type, which may be a built-in type in the programming language or an ADT. A mathematical theory sophisticated enough to handle the array specification is an additional complication [10]. Unless this theory contains special operators that hide them, quantifiers are involved in the specification; to simplify automated verification in the presence of existential quantifiers, ghost (or adjunct) variables may be needed in the specification and/or implementation. This code also involves at least two types: the numerical index type and the array type, and possibly a separate entry type. A recommended candidate for the incorrect version of the code is any close cousin of the Java binary search function whose defect was pointed out recently in a Google blog "after lying in wait for nine years or so" [11]. The numerical index type for the array must be considered bounded to manifest the defect in that code.

Variations: An array may have fixed bounds, or they may be dynamically adjustable. If the type of the array elements is not fixed but rather is a parameter to the specification(s) and code, then additional issues arise here, before benchmark #3 brings them front and center. In particular, the actual entry type might be an ADT itself; the specification and computation of the ordering among entries and equality of entries become interesting.

2.3 Benchmark #3: Sorting a Queue

Problem Requirements: Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.

Issues: Dealing with a generic collection type is one new issue here. In addition, a mathematical theory suitable for specifying and verifying queue behavior may be needed—perhaps different than that used for arrays in benchmark #2 [12]. Parameterizing the sort operation to account for the specification and computation of the ordering among entries is central. Implementations that involve nested loops may require ghost variables to simplify writing loop invariants.

Variations: Some variations that delay the inevitable generic type issues might keep the problem easier, while leaving it incrementally further along the path to the goal than benchmark #2. For example, even if the queue entries are a fixed type with a fixed ordering for sorting, the mathematical definitions and/or new theory to address even this simplified version of the benchmark might be troublesome.

2.4 Benchmark #4: Layered Implementation of a Map ADT

Problem Requirements: Verify an implementation of a generic map ADT, where the data representation is layered on other built-in types and/or ADTs.

Issues: This may be viewed as "recasting" benchmark #2 as an ADT. The issues of proof of correctness of data representation [13] are now involved, including representation invariants and abstraction relations [14]. There are many versions of "map" behavior, including impoverished ones that obscure important issues. The map designed for this benchmark should be as useful to clients as, say, maps in the *java.util* package. If any of the map operations involves relational behavior, a careful definition of correctness might be subtle.

Variations: A simple data representation such as a queue with linear search as the primary algorithm is an obvious starting point. There are many other data structures and algorithms with better performance that are more realistic (e.g., hash tables of various ilks, binary search trees with or without balancing). Each of these alternatives might involve specifying other interesting ADTs as the basis for the data representation. Additional issues might arise from this exercise.

2.5 Benchmark #5: Linked-List Implementation of a Queue ADT

Problem Requirements: Verify an implementation of the queue type specified for benchmark #3, using a linked data structure for the representation.

Issues: Pointers/references may be either built-in types in the programming language or user-defined types. Specifications for their behavior, and verification of linked data structures that use them, raise many interesting questions [15,16,17]. Most obvious is that control of aliasing becomes an issue. If this is not addressed carefully, retaining modularity of verification might be difficult.

Variations: A linked data structure might be encoded in an array with integer indices, so issues related to linked data structures are separated from issues related to language-defined pointers/references as a way of introducing indirection. If memory is allocated dynamically, then memory might be assumed to be unbounded or bounded, the latter raising the specter of memory allocation that does not always succeed. The queues themselves might be specified to be unbounded, or bounded in various ways (e.g., uniformly bounded so each queue has the same maximum length, or communally bounded so the sum of the lengths of all queues is bounded).

2.6 Benchmark #6: Iterators

Problem Requirements: Verify a client program that uses an iterator for some collection type, as well as an implementation of the iterator.

Issues: There are many proposed designs for iterators, each of which raises a number of interesting specification and verification issues that involve coupling between an underlying collection type and an iterator for it. Specifying iterators (though not verifying anything about them) was a challenge problem issued for the *SAVCBS Workshop* in 2006 [18,19].

Variations: Iterators may be active or passive [20]. Each design has its own set of problems [21]. A passive iterator, where an operation is passed to the iterator and the iterator applies it to each entry, starts to raise issues similar to those involved in callbacks. None of the benchmarks proposed here deals with callbacks.

2.7 Benchmark #7: Input/Output Streams

Problem Requirements: Specify simple input and output capabilities such as character input streams and output streams (with the flavor of C++ streams, for example). Verify an application program that uses them in conjunction with one of the components from the earlier benchmarks.

Issues: Technically, modeling input and output might involve concurrent processes operating alongside an application program. It is far from clear this is the best way to view the situation, though, and the modeling and specification issues that arise could take a solution in any of a number of intriguing directions.

Variations: A version of I/O streams involving "standard input and output" streams that can be redirected from/to files is one way to keep I/O in a program separate from the notion of a file system. A variation that includes the possibility of opening I/O streams to files by their filenames entails coupling with the file system, raising a host of issues related to persistent data. Going this direction might involve connections with a challenge problem for the *ABZ 2008 Conference*, i.e., specification and verification of the POSIX file-store interface of Unix [1]. A variation that allows I/O streams to serve as an interprocess communication mechanism raises yet other issues that lead into the full gamut of concurrency questions.

2.8 Benchmark #8: An Integrated Application

Problem Requirements: Verify an application program with a concisely stated set of requirements, where the particular solution relies on integration of at least a few of the previous benchmarks. For example, verify an application program that does the following: Given input containing a series (in arbitrary order) of terms and their definitions, output an HTML glossary that presents all the terms and their definitions, with (a) the terms in alphabetical order, and (b) a hyperlink from each term that occurs in any definition to that term's location in the glossary.

Issues: This benchmark adds the issues involved in composing a number of user-defined types and operations. Code for a glossary generator, for instance, might use the sorting operation from benchmark #3, the map type from benchmark #4, I/O streams from benchmark #7, and perhaps others. Modularity implies that implementations of all the components being integrated need not be verified, too, but of course they must be specified in order for modular verification of the application program to proceed.

Variations: If the I/O stream specifications from benchmark #7 include access to the file system, then an interesting variation of the glossary generator requirements is to structure the glossary as a single index page plus a set of HTML pages, one per term, with hyperlinks between the pages.

3 Two Solutions to Part One of Benchmark #1

Two solutions to the "addition by repeated incrementing" part of benchmark #1 are presented as guides to an appropriate level of detail for a proposed solution. These solutions use syntactically slightly different, though semantically identical, dialects of the RESOLVE language [22,23,24]. The first solution is recursive and uses a VC generator under development at Clemson; the VCs are proved using the automated proof assistant Isabelle [25]. For variety, the second solution is iterative and uses a different set of tools under development at Ohio State to generate and prove the VCs. The benchmark requirement to show a "slightly" incorrect version is not illustrated here, but such code is available at the respective web sites mentioned below.

3.1 Recursive Procedure for Addition by Repeated Incrementing

For more details, see:

`http://www.cs.clemson.edu/~resolve/benchmarks`

Input Specification and Code for Verification: The specification and implementation of the `Add_to` operation are given with access to a specification of an `Integer` type and operations. Parameter modes, only **`updates`** and **`evaluates`** in this code, are specification constructs. The **`updates`** mode indicates the parameter may be modified in any way permitted by the **`ensures`** clause. The **`evaluates`** mode allows an expression to be passed as the corresponding argument, i.e., it indicates a "value" parameter. In an **`ensures`** clause, the prefix # for a formal parameter denotes the incoming value. Recursive code is annotated with a progress metric using the keyword **`decreasing`**.

```
Operation Add_to(updates i:Integer; evaluates j:Integer);
    requires min_int <= i + j and i + j <= max_int and j >= 0;
    ensures  i = #i + j;
```

```
Procedure Add_to(updates i:Integer; evaluates j:Integer);
    decreasing |j|;
        Var z: Integer;
        If (not Is_Zero(j)) then
            Increment (i);
            Decrement (j);
            Add_to (i, j);
        end;
end Add_to;
```

In RESOLVE, all types (including those built-in to most languages) are specified, used, and verified in a uniform way. `Integer` is specified in `Integer_Template` shown below, where `Z` denotes mathematical integers; this is defined in a mathematical module `Integer_Theory` (not shown) along with mathematical notations such as `0`, `+` , and `-`. This specification **`defines`** two constants `min_int` and `max_int`. Every `Integer` variable is constrained to be within these bounds, and is initially 0.

```
Concept Integer_Template;
    uses Integer_Theory, Std_Boolean_Fac;

    Defines min_int: Z;
    Defines max_int: Z;
    Constraint min_int <= 0 and 0 < max_int;

    Type Family Integer is modeled by Z;
        exemplar i;
        constraint min_int <= i and i <= max_int;
        initialization ensures i = 0;

    Operation Is_Zero(evaluates i: Integer): Boolean;
        ensures  Is_Zero = ( i = 0 );

    Operation Increment(updates i: Integer);
        requires i + 1 <= max_int;
        ensures  i = #i + 1;

    Operation Decrement(updates i: Integer);
        requires min_int <= i - 1;
        ensures  i = #i - 1;
     …
end Integer_Template;
```

Verification Conditions: VCs are generated using a tool that implements the proof rules described in [26,27]. Briefly, there is a VC for each state in the program where the next statement involves establishing a precondition for the next operation, a loop invariant or progress metric, or the postcondition for the operation being verified. The 8 VCs are shown below. All variables are of type Z. Each VC is independent of the others, and consists of a goal that needs to be proved (below the line) using several hypotheses (above the line). For example, VC 3 arises from the ordinal-valued **decreasing** clause to show termination.

```
(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________________
((i + 1) <= max_int)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________________
(min_int <= (j - 1))

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________________
(|(j - 1)| < |j|)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________________
```

```
(min_int <= ((i + 1) + (j - 1)))

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________
(((i + 1) + (j - 1)) <= max_int)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________
((j - 1) >= 0)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________
((i + 1) + (j - 1)) = (i + j)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and j = 0)))
_____________________________
i = (i + j)
```

Verification Results: Isabelle-friendly versions of the VCs are generated to provide proofs. Proofs of all VCs are completed automatically by Isabelle with "apply force".

3.2 Iterative Procedure for Addition by Repeated Incrementing

For more details, see:

`http://www.cse.ohio-state.edu/rsrg/benchmark-results/add-mult`

Input Specification and Code for Verification: The numbers in this version of the problem are unbounded natural numbers, which also are not built-in. The parameter mode **`restores`** means there is no net change to the parameter's value, i.e., it means implicit addition of a conjunct such as `m = #m` to the **`ensures`** clause.

```
procedure Add (updates n: Natural, restores m: Natural)
    ensures
        n = #n + m
```

```
procedure Add (updates n: Natural, restores m: Natural)
    variable k, z: Natural
    loop
        maintains n + m = #n + #m  and  k + m = #k + #m  and
                  z = 0
        decreases m
    while not AreEqual (m, z) do
        Increment (n)
        Increment (k)
        Decrement (m)
    end loop
    m :=: k
end Add
```

```
contract UnboundedNaturalFacility

    math subtype NATURALMODEL is integer
        exemplar n
        constraint n >= 0

    type Natural is modeled by NATURALMODEL
        exemplar n
        initialization ensures n = 0

    procedure Increment (updates n: Natural)
        ensures  n = #n + 1

    procedure Decrement (updates n: Natural)
        requires n > 0
        ensures  n = #n - 1

    function AreEqual (restores m: Natural,
                       restores n: Natural): control
        ensures  AreEqual = (m = n)

    function IsGreater (restores m: Natural,
                        restores n: Natural): control
        ensures  IsGreater = (m > n)

    function Replica (restores n: Natural): Natural
        ensures  Replica = n

end UnboundedNaturalFacility
```

The algorithm for addition is the obvious iterative one. The only possibly unusual part of the code is the swap statement after the loop, which is a RESOLVE staple that replaces assignment as the built-in data movement operator.

Verification Conditions: VCs are generated in a locally defined XML format, using a tool we have developed that implements the proof rules described in [28,29]. This XML is then translated into human-readable unicode format and into input for Isabelle [25], and also is piped directly into SplitDecision (see below).

As in Section 3.1, there is a VC for each state in the program where the next statement involves establishing a pre-condition for the next operation, a loop invariant or progress metric, or the post-condition for the operation being verified. However, the proof rules in this version involve profligate use of mathematical variables: one for each program variable in each program state (between consecutive statements). So, for example, x_i stands for the value of program variable x in state i. In order to relieve the burden of automatic case analysis by the back-end prover, the VC generator divides each VC into cases based on the path structure of the code. In addition, it does a number of simplifications that do not rely on knowledge of particular mathematical domains, but rather depend only on general logical properties (e.g., substitution of equals). There are too many VCs to show here (12 total, the largest having 23 hypotheses before any simplification and 6 afterward); a representative VC is reproduced below both in a human-readable form and in the form generated for Isabelle. This VC is from state 5, just after the `Decrement` call inside the loop, and arises because we must show that (each clause of) the loop invariant holds at the end of the loop body.

```
Lemma #6 (state index: 5)
————————————————————————
     n_0 ≥ 0
∧    m_0 ≥ 0
∧    m_2 ≠ 0
∧    n_2 + m_2 = n_0 + m_0
∧    k_2 + m_2 = 0 + m_0
∧    m_2 > 0
     ————————————————————
⇒    n_2 + 1 + (m_2 — 1) = n_0 + m_0
```

```
lemma 6:
"[|
     (n_0::int) >= 0 ;
     (m_0::int) >= 0 ;
     ~(m_2::int) = 0 ;
     (n_2::int) + m_2 = n_0 + m_0 ;
     (k_2::int) + m_2 = 0 + m_0 ;
     m_2 > 0
  |]
  ==>
     n_2 + 1 + (m_2 - 1) = n_0 + m_0"
apply (((simp only: simp_thms),clarify?)+)?
apply (force+)?
done
```

Verification Results: SplitDecision is a simplifier we are developing that uses mathematical properties involving theories used in RESOLVE specifications to further simplify VCs, based on knowledge of predicate calculus with equality and using special-purpose decision procedures we have designed for relevant fragments of the mathematical theories of integers and strings (so far). Proofs (simplifications all the way to "true") of all the VCs for this problem are completed automatically and quickly by both Isabelle and SplitDecision.

4 Discussion and Conclusions

As others also have suggested, albeit without elaborating a set of benchmarks [30], it will be helpful to have several reachable milestones along the road to the VSI goal of routinely verifying real-world software systems. In fact, one of the reviewers of this paper summarized the point very clearly: "A software verification system failing the benchmark suite can hardly be ready for proving the correctness of serious software." The benchmarks and methodology proposed here for documenting incremental progress constitute a first step in defining specific milestones and capabilities.

While we have focused on code verification in our sample solutions, it is clear that the benchmarks also could be used as suitable targets for illustrating issues in specification, generating internal assertions, teaching formal verification, or for proving certain classes of verification conditions. An important feature toward this end is that the proposed benchmarks are stated in terms of brief requirements statements. Why not start with "reference formal specifications"? The software community still does not

agree about how formal specifications should be written, or even about the design and intended behavior of the simplest components; if there is any doubt about this, compare the JML [31] and RESOLVE/C++ [32] specifications for a stack component. It is, therefore, unlikely that a verification system that is required by benchmark statements to adopt someone else's software designs and specifications would ever be tried—let alone succeed—on such benchmarks. A similar conclusion applies to the programming language used for implementations. While it will be exciting if and when the VSI effort ultimately succeeds in creating tools that can verify software that is not designed for verification, e.g., open-source software, we submit this is not the place to start. The hope is that the proposed benchmarks *will not appear daunting* to those who claim to have automated verification systems for proving total correctness, and that this very feature will encourage everyone to participate and thereby facilitate comparisons among different approaches along many dimensions of the problem.

We trust the VSI community also will find it an acceptable challenge to improve upon and augment this set of benchmarks to illustrate other important issues in relatively simple and incremental ways. For example, if there is widespread interest in moving in such directions, then those interested in exploring them should devise additional benchmarks—meeting the general requirements outlined at the beginning of Section 2—to address verification of open-source software, software specified in certain styles and languages, software written in specific commercial languages or with specific features, etc.

Acknowledgments

The authors appreciate the contributions of Jeremy Avigad, Harvey M. Friedman (whose decision procedure for strings with some restrictions is used in SplitDecision), Greg Kulczycki, Bill Ogden, and Anna Wolf. This work is supported in part by the National Science Foundation under grants DMS-0701187 and DMS-0701260.

References

1. ABZ call for papers on the POSIX pilot project in the grand challenge (viewed April 27, 2008), http://www.abz2008.org
2. Pacemaker formal methods challenge (viewed April 27, 2008), http://www.cas.mcmaster.ca/sqrl/pacemaker.htm
3. Stepney, S., Cooper, D., Woodcock, J.: An electronic purse: specification, refinement, and proof. PRG Technical Monograph PRG-126 (2000), 231 pages (viewed April 27, 2008), http://web2.comlab.ox.ac.uk/oucl/publications/monos/prg-126.html
4. Woodcock, J., Banach, R.: The verification grand challenge. Journal of Universal Computer Science 13(5), 661–668 (2007)
5. Schmitt, P.H., Tonin, I.: Verifying the Mondex case study. In: Fifth IEEE Intl. Conf. on Software Engineering and Formal Methods, pp. 47–58. IEEE, Los Alamitos (2007)
6. Haneberg, D., Schellhorn, G., Grandy, H., Reif, W.: Verification of Mondex electronic purses with KIV: from transactions to a security protocol. Formal Aspects of Computing 20(1), 41–59 (2007)

7. The POPLmark challenge (viewed April 27, 2008), http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/index.php?title=The_POPLmark_Challenge
8. Bicarregui, J.C., Hoare, C.A.R., Woodcock, J.C.P.: The verified software repository: a step towards the verifying compiler. Formal Aspects of Computing 18(2), 143–151 (2006)
9. Ancient Egyptian multiplication (viewed April 27, 2008), http://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication
10. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer, Heidelberg (2007)
11. Bloch, J.: Extra, extra – read all about it: nearly all binary searches and mergesorts are broken (2006) (viewed 27 April 2008), http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
12. Zhang, T., Sipma, H.B., Manna, Z.: Decision procedures for queues with integer constraints. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 225–237. Springer, Heidelberg (2005)
13. Hoare, C.A.R.: Proof of correctness of data representations. Acta. Inf. 1(4), 271–281 (1972)
14. Sitaraman, M., Weide, B.W., Ogden, W.F.: On the practical need for abstraction relations to verify abstract data type representations. IEEE Transactions on Software Engineering 23(3), 157–170 (1997)
15. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proc. 17th Annual IEEE Symp. on Logic in Computer Science, pp. 55–74. IEEE, Los Alamitos (2002)
16. Kulczycki, G.: Direct Reasoning. Dept. of Computer Science, Ph.D. thesis, ClemsonUniversity, Clemson, SC (2004)
17. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: ACM Conference on Programming Language Design and Implementation, pp. 349–361. ACM Press, New York (2008)
18. Challenge problem: iterator specification (viewed April 27, 2008), http://www.eecs.ucf.edu/~leavens/SAVCBS/2006/challenge.shtml
19. Leavens, G., (ed.): SAVCBS 2006 Proceedings: Specification and Verification of Component-Based Systems (viewed April 27, 2008), http://www.eecs.ucf.edu/~leavens/SAVCBS/2006/SAVCBS06-proceedings.pdf
20. Booch, G.: Software components with Ada. Benjamin Cummings, Redwood City, CA (1987)
21. Weide, B.W., Edwards, S.H., Harms, D.E., Lamb, D.A.: Design and specification of iterators using the swapping paradigm. IEEE Transactions on Software Engineering 20(8), 631–643 (1994)
22. Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W.: Specifying components in RESOLVE. Software Engineering Notes 19(4), 29–39 (1994)
23. Bucci, P., Hollingsworth, J.E., Krone, J., Weide, B.W.: Implementing components in RESOLVE. Software Engineering Notes 19(4), 40–52 (1994)
24. Kulczycki, G., Sitaraman, M., Yasmin, N., Roche, K.: Formal specification. In: Encyc. of Computer Science and Engineering. John Wiley & Sons, Chichester (to appear, 2008)
25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
26. Krone, J.: The Role of Verification in Software Reusability. Ph.D. dissertation, Dept. of Comp. and Inf. Science, Ohio State Univ., Columbus, OH (1988)
27. Harton, H., Krone, J., Sitaraman, M.: Formal program verification. In: Encyc. of Computer Science and Engineering. John Wiley & Sons, Chichester (to appear, 2008)

28. Heym, W.D.: Computer Program Verification: Improvements for Human Reasoning. Ph.D. dissertation, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH (1995)
29. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S., Hollingsworth, J.E.: Reasoning about software-component behavior. In: Frakes, W.B. (ed.) ICSR 2000. LNCS, vol. 1844, pp. 266–283. Springer, Heidelberg (2000)
30. Zhang, J.: Program verification in the small. In: Proc. First Asian Working Conf. on Verified Software, UNU/IIST Report #348, pp. 83–84 (2006)
31. JML BoundedStackInterface (viewed July 15, 2008), http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/org/jmlspecs/samples/stacks/BoundedStackInterface.html
32. RESOLVE/C++ Catalog, AT/Stack/Kernel.h (viewed July 15, 2008), http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/AT/Stack/Kernel.html

Verified Protection Model of the seL4 Microkernel

Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone

NICTA* and University of New South Wales,
Sydney, Australia
{dhammika.elkaduwe,gerwin.klein,kevin.elphinstone}@nicta.com.au

Abstract. This paper presents a machine-checked high-level security analysis of seL4—an evolution of the L4 kernel series targeted to secure, embedded devices. We provide an abstract specification of the seL4 access control system together with a formal proof that shows how confined subsystems can be enforced. All proofs and specifications in this paper are developed in the interactive theorem prover Isabelle/HOL.

1 Introduction

We present a machine-checked high-level security analysis of seL4 [5,6], an evolution of the L4 kernel series [12] targeted to secure, embedded devices.

It does not need to be argued that embedded systems have become an integral part of our lives. They are increasingly deployed in safety- and mission-critical scenarios. Even relatively simple devices like mobile phones feature millions of lines of software, installed for various purposes, with varying degrees of assurance, with diverse resource requirements, and developed on a tight resource budget. They feature untrusted third-party software components, applications, and even whole operating systems (such as Linux) that can be installed by the manufacturer, suppliers and the end user.

Microkernels are a promising approach with renewed industry interest to improving the security and robustness of such devices. The success of this approach depends to a large degree on the microkernel's ability to provide strong isolation guarantees between components—misbehaviour of a component should be confined to that component only.

In this paper, we analyse the seL4 kernel primitives and affirm that they are sufficient to enforce isolation. Moreover, we demonstrate through examples that the restrictions imposed are pragmatic: The mechanisms can be used in practice.

The main contributions of this work are: (a) to our knowledge, the first machine-checked specification and first machine-checked proof of a take-grant [13] (TG) model, (b) making the graph diagram notation that is used for TG

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 99–114, 2008.

analysis in the literature fully precise, (c) extending classical TG in the seL4 protection model by using it to control the in-kernel physical memory consumption of applications, thereby making it feasible to reason about and control the physical memory consumption of applications based on the distribution of authority, and (d) applying the model directly to the seL4 kernel.

All formal definitions, theorems, and examples in this paper are machine-checked in the theorem prover Isabelle/HOL [15].

This paper is one of the steps towards our longer-term goal of full operating systems verification. We aim to formally connect the security proof presented here with the actual kernel implementation in C. All of the seL4 operations map to one or more of the operations presented in this paper. Many of them, e.g., basic thread management, translate to no-op, but the authority-relevant operations have a direct representation in the model presented here. We have so far connected an abstract operational model of seL4 with a precise executable one [3]. We are concurrently working on a formal refinement proof between the security model of this paper and this abstract model and also on the refinement proof between the executable model and the C implementation [7,8]. The security specification is ca. 300loc, the abstract one about 3kloc, the executable one 7kloc, the C code 10kloc, and the refinement proof between abstract and executable specification 100kloc. In this paper, we focus on the tip of this iceberg: the high-level aspects of seL4 security.

2 The seL4 Microkernel

The seL4 (*secure embedded L4*) kernel is an evolution of the L4 microkernel. It employes a capability [4] based protection system that is inspired by early hardware-based capability machines such as CAP where capabilities control access to physical memory, by the KeyKOS and EROS systems [9,19] with their controls on dissemination of capabilities, and by the take-grant model [13]. In this section, we provide an overview of the relevant parts of the seL4 kernel. A detailed exposition can be found elsewhere [5,6,14].

Similar to L4, the seL4 microkernel provides three basic abstractions: threads, address spaces and inter-process communication (IPC). In addition, seL4 introduces the concept of *untyped memory*, which represents a region of currently unused physical memory.

All kernel abstractions and services are provided via named, first-class kernel objects. Authority over these objects is conferred via capabilities. Any seL4 system call is a capability invocation. System call arguments can either be data or other capabilities — authorised users obtain kernel services by invoking capabilities.

A parent capability to untyped memory can be refined into child capabilities to smaller untyped memory blocks or into other kernel objects via the *retype* operation. The creator can then delegate all or part of the authority it possesses over the object to one or more of its clients. This is done by *granting* the client a capability to the object with possibly diminished access rights.

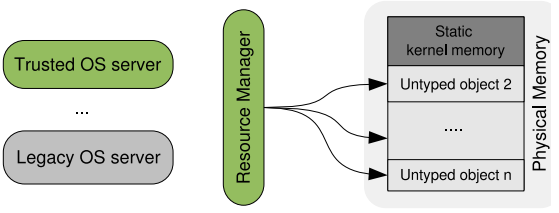


Fig. 1. Sample System Configuration

An important part of the seL4 design is that all memory—be it the memory directly used by an application (e.g. memory frames) or indirectly in the kernel (e.g. page tables), is fully accounted for by capabilities.

At boot time, seL4 preallocates all memory required for the kernel to run—space for kernel code, data, and kernel stack. As shown in Fig. 1 the remainder of memory is divided into untyped memory (UM) objects. The rounded boxes in Fig. 1 stand for threads (OS servers, resource manager), the arrows stand for capabilities, and the box on the right hand side represents available physical memory. The initial user-level thread, the resource manager, has full authority over the UM objects; it is responsible for enforcing a suitable resource management policy, and for bootstrapping the rest of the system. Behaviour of any application, the legacy OS server in Fig. 1 for instance, is constrained by the capabilities the resource manager grants it. Constraining these capabilities appropriately isolates the legacy OS server from the rest of the system.

By this flexible, delegatable, but still precise accounting of all physical memory through capabilities in seL4, the question of partitioning hardware resources becomes a question of capability distribution only. The next sections show how capability distribution is modelled and controlled.

3 The seL4 Protection Model

In the remainder of this paper, we formally analyse the access control model of the seL4 kernel. Our goal is to show that it is feasible to implement *isolated subsystems* using seL4 mechanisms. An isolated subsystem can be viewed as a collection of processes or *entities* encapsulated in such a way that authority can neither get in nor out. This also means that the subsystem cannot gain access to any additional physical memory at any time in the future and is thus strongly spatially separated from the rest of the system.

We start with an example of our requirements, which we carry forward in the discussion below. Assume there are n distinct subsystems in our system, namely $ss_1, ss_2 \ldots ss_n$ (see Fig. 2). Each subsystem may contain one or more processes and these processes may have access to other resources or processes. In Fig. 2 and the following figures, we use shaded rounded boxes to represent processes and shaded circles to denote resources. We draw arrows between these

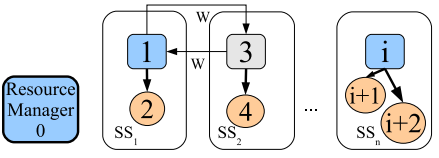


Fig. 2. Isolated Subsystems

numbered entities to denote capabilities. The larger rounded boxes mark subsystem boundaries.

The resource manager responsible for setting up these subsystems would like to guarantee that any given subsystem, say ss_i, cannot exceed the authority explicitly given to it, and with that, the amount of physical memory and communication channels[1]. In other words, after providing all subsystems with their initial capabilities, the resource manager would like to guarantee that no entity within ss_i can obtain capabilities to an entity in another subsystem unless these capabilities are already already present in ss_i (possibly in another entity of ss_i). Note that we are not restricting the flow of authority within a subsystem—capabilities can flow freely within a subsystem, but are not allowed to cross the subsystem boundary.

Our interest is in *mandatory* isolation: that is, in showing that a subsystem *cannot* acquire additional capabilities, rather than *it can but does not*.

The initial state in which subsystems start executing (state s_1), is created by and hence under the strict control of the resource manager. Derived states are subsequent states that are affected by the execution of subsystems. To make strong guarantees, we need to show that subsystem boundaries are not violated in any derived state. In the formal analysis below, we identify a set of invariants the resource manager can enforce on s_1, such that there is no sequence of commands that can violate any subsystem boundary.

We begin the analysis by formalising the access control model.

3.1 Formalisation

The system state consists of a collection of kernel objects. We do not make the usual distinction between active subjects and passive objects. Instead, we collectively call them entities. Entities are identified by their unique address which we model as natural numbers: `entity_id::nat`. An entity contains a set of capabilities which we define below, and has no additional authority beyond what it possesses as capabilities:

record `entity` = `caps :: cap set`

A capability is a record with two fields: (a) an identifier which names an entity and (b) a set of access rights which defines the operations the holder is authorised to perform.

[1] We do not consider covert timing channels in this paper.

record *cap* = *entity :: entity_id*
rights :: rights set

where

datatype *rights* = *R* | *W* | *G* | *C*

The data type *rights* defines the four primitive access rights in our model. Out of these rights, *R* and *W* have the obvious meaning. They authorise reading and writing of information. The *G* right is sufficient authority to grant a capability to another entity. The *C* right models the behaviour of untyped memory objects. It confers the authority to create new entities. We use the term *allRights* to denote the set of all the access rights; formally *allRights* ≡ *{R, W, G, C}*.

The state of the whole system consists of two fields:

record *state* = *heap :: entity_id* ⇒ *entity*
next_id :: entity_id

The component *heap* stores the entities of the system, it maps entity addresses (*entity_id*) to entities. The *next_id* is the next free slot for placing an entity without overlapping with any existing one.[2] This setup allows a simple test to determine the existence of an *entity_id*:

is_entity :: state ⇒ *entity_id* ⇒ *bool*
is_entity s e ≡ *e* < *next_id s*

This test is not present in the kernel implementation itself. In the implementation, the existence of a capability in the system implies the existence of the entity. The same is true in our abstract model for well-formed states: the entities stored in *heap* contain capabilities, which again contain references to other entities. In any run of the system, these references should only point to existing entities. We call such system states *sane*:

sane s ≡ (∀ *c*∈*all_caps s. is_entity s* (*entity c*)) ∧
(∀ *e*. ¬ *is_entity s e* ⟶ *caps_of s e* = ∅)

where *caps_of s r* is the set of all capabilities contained in the entity at address *r* in state *s*, formally *caps_of s r* ≡ *caps* (*heap s r*)[3], and *all_caps s* is the union of the capabilities over all entities in the state *s*, formally *all_caps s* ≡ ⋃*e caps_of s e*.

Next, we introduce the operational semantics of our model, captured in the function *step'*, shown in Fig. 3. The first argument to *step'* is the operation to perform. The second argument is the current system state and the result is the mutated state after performing the specified operation on the current state.

The first argument of each operation is the entity initiating the operation. The second argument is the capability being invoked. The third argument for

[2] An alternative to this model would be to use a partial function for the heap. We found working with a total function and an explicit, separate domain slightly more convenient in this case.

[3] *heap s* selects the field *heap* from record *s*, (|*caps* = ∅|) is the record of type *entity* containing the empty cap set, and *s*(|*heap* := *h'*|) is the record *s* where field *heap* is replaced by *h'*.

```
step' :: sysOPs ⇒ state ⇒ state

step' (Create e c₁ c₂) s  = let newObj = (|caps = ∅|);
                                newCap = (|entity = next_id s, rights = allRights|);
                                newSrc = (|caps = {newCap} ∪ caps_of s (entity c₂)|)
                            in (|heap = (heap s)(next_id s := newObj, entity c₂ := newSrc),
                                 next_id = next_id s + 1|)
step' (Grant e c₁ c₂ r) s = s(|heap := (heap s)
                                (entity c₁ :=
                                   (|caps = {diminish c₂ r} ∪ caps_of s (entity c₁)|))|)
step' (Remove e c₁ c₂) s  = removeOperation e c₁ c₂ s
step' (Revoke e c) s      = foldr (removeCaps e) (cdt s c) s
step' _ s                 = s
```

where

```
removeOperation e c₁ c₂ s ≡ s
(|heap := (heap s)(entity c₁ := (|caps = caps_of s (entity c₁) - {c₂}|))|)

removeCaps e (c, cs) s ≡ foldr (removeOperation e c) cs s

cdt :: state ⇒ cap ⇒ (cap × cap list) list
```

Fig. 3. Single step execution

Create points to the destination entity for the new capability, for *Grant* it is the capability that is transported and for *Remove* the capability that is removed. The fourth argument to *Grant* is a mask for the rights of the transported capability.

There are three additional operations in the model that are matched in the last equation of *step'*: *NoOp*, *Read*, and *Write*.

As the name implies, *NoOp* does nothing, and usually is not present in traditional abstract system models. However, it is included here, because some of the operations that exist in the seL4 kernel API will not be observable on this abstract level and thus can only be mapped to *NoOp*. An example is sending a non-blocking message to a thread not willing to accept, which will result in a dropped message. In fact, neither *Read* nor *Write* change the abstract system state either. We include them in this model, because they have preconditions that are observable on this level and thus might be interesting for later analysis.

The *Create* operation allocates a new entity in the system heap, creates a new capability to the new entity with full authority and places this new capability in the destination entity pointed by c_2. The operation consumes resources in terms of creating the new entity in the heap. So, the subject initiating this call is required to invoke an untyped capability c_1.[4] Placing the new capability does not consume additional resources. Moreover, when performed on a *sane* state, the create operation guarantees: (a) the new entity will not overlap with any of the existing ones and, (b) no capability in the current state will be pointing to the heap location of the new entity.

The *Grant* operation adds a capability to the entity pointed to by c_1. However, unlike *Create*, the added capability is a diminished copy of the existing capability c_2. The *diminish* function reduces access rights according to the mask specified

[4] The model presented here does not take into account that memory is limited. For refinement, we introduce non-deterministic failure for operations like *Create* to mimic the real behaviour, but do not specify exactly when memory runs out.

```
legal :: sysOPs ⇒ state ⇒ bool

legal (NoOp e) s           = is_entity s e
legal (Read e c) s         = is_entity s e ∧ c ∈ caps_of s e ∧ R ∈ rights c
legal (Write e c) s        = is_entity s e ∧ c ∈ caps_of s e ∧ W ∈ rights c
legal (Create e c1 c2) s   = is_entity s e ∧
                             {c1, c2} ⊆ caps_of s e ∧ G ∈ rights c2 ∧ C ∈ rights c1
legal (Grant e c1 c2 r) s = is_entity s e ∧ {c1, c2} ⊆ caps_of s e ∧ G ∈ rights c1
legal (Remove e c1 c2) s   = is_entity s e ∧ c1 ∈ caps_of s e
legal (Revoke e c) s       = is_entity s e ∧ c ∈ caps_of s e
```

Fig. 4. Preconditions for executing operations

in the `Grant` operation, facilitating an entity to propagate a subset of its own authority to the receiver.

```
diminish cap r ≡ cap(|rights := rights cap ∩ r|)
```

Both the `Remove` and `Revoke` operations remove capabilities: in the former case from the entity pointed to by c_1 and in the latter case from a whole system. To facilitate `Revoke`, the seL4 kernel internally tracks in a capability derivation tree (*cdt*) [6] how capabilities are derived from one another with create and grant operations. We do not model the cdt explicitly at this level, instead we assume the existence of a function `cdt` that returns for the current system state and the capability to be revoked, a list that describes which capabilities are to be removed from which entities. Given this list, the revoke operation is then just a repeated call of `Remove`.

Any operation is allowed only under certain preconditions, encoded by `legal`, as defined in Fig. 4. The definition of `legal` firstly checks if the entity initiating the operation exists in that system state. Secondly, all the capabilities specified in the operation should be in the entity's possession at that state. Finally, the capabilities specified should have at least the appropriate permissions.

The single step execution function `step` first checks whether the operation is legal in the current state and if so, calls `step'`.

```
step :: sysOPs ⇒ state ⇒ state
step cmd s = (if legal cmd s then step' cmd s else s)
```

Executing a list of system operations is then just repetition of `step`. Note that the list of commands is read from right to left here.

```
execute :: sysOPs list ⇒ state ⇒ state
execute = foldr step
```

After the kernel bootstraps itself, it creates state s_0 with one entity; the resource manager, which possesses full rights to itself. In the concrete kernel, the initial state is slightly more complex, containing the resource manager thread and a number of separate untyped capabilities to cover all available memory. We have folded these here into $s_0 \equiv$ `(|heap = [0 ↦ {allCap 0}], next_id = 1|)`. The notation `[0 ↦ {allCap 0}]` stands for an empty heap where position `0` is overwritten with an object that has `{allCap 0}` as its capability set, where

allCap e ≡ (|*entity = e, rights = allRights*|). Note that this initial state is sane, and that execution preserves sanity. Formally:

$$sane\ s \Longrightarrow sane\ (execute\ cmds\ s)$$

Thus, all states considered in the analysis below are sane.

4 Mandatory Isolation of Components

In this section, we show that it is feasible to implement *isolated subsystems* in seL4. A subsystem is merely a set of entities related to one another in a certain way, which we define later. By isolated we mean that none of the entities in the subsystem ss_1 will gain access to a capability to an entity of another subsystem ss_2 if that authority is not already present in ss_1. If the authority is already present, then we show that it cannot be increased. Subsystems can grow over time and the statement also includes entities that currently do not exist yet.

Our focus is on authority propagation or capability *leakages* from one subsystem to another. We start by considering two entities: e_x and e_y. We write $s \vdash e_x \rightarrow e_y$ to denote that entity e_x has the ability to leak authority to entity e_y in state s.

leak :: state ⇒ entity_id ⇒ entity_id ⇒ bool
s ⊢ e_x → e_y ≡ *gCap* e_y *:< caps_of s* e_x
where
gCap e ≡ (|*entity = e, rights = {G}*|), and
c :< S ≡ ∃ *c'*∈*S. entity c = entity c' ∧ rights c* ⊆ *rights c'*

where the notation *c :< S* reads *the capability set S provides as least as much authority as the capability c*. Based on the operational semantics of our model, there are two operations that may create such a leak—*Create* and *Grant*, and these are legal only if the entity initiating the operation has a capability to the entity under consideration with at least grant (*G*) authority. We therefore define a subsystem as a set of entities connected by grant capabilities. From the analysis in Sect. 4.1 we identify a property that is preserved by *step* and hence by *execute*, which can be used to decide whether an entity will be able to leak to another in the future. Sect. 4.2 then extends this result to show how isolated subsystems can be implemented using seL4. Sect. 4.3 illustrates how isolated subsystems are implement in our abstract model, together with a discussion on how we realised this in the concrete system. The proof is 1200 lines of Isabelle script. We show here only the essential lemmas and their intuition.

4.1 Conditions for Authority Propagation

The invariant property of the system relating to propagation of authority is the symmetric, reflexive and transitive closure over the *leak* relation. Occasionally, the symmetric closure alone is useful. We call it *connected* and write $s \vdash e_x \leftrightarrow e_y$.

The intuition behind this invariant is the following. We are looking at grant capabilities only, because these are the only ones that can disseminate authority. We need the transitive closure, because we are looking at an arbitrary number of execution steps. We need the symmetric closure, because as soon as there is one entity in the transitive closure that has a grant capability to itself, it can use this capability to invert grant arcs. Given the transitive and symmetric part, the reflexiveness follows.

Next, we analyse the effect of each operation on the `connected` relation. Obviously, `NoOp`, `Read` and `Write` do not modify the capability distribution and therefore cannot affect `connected`. Similarly, `Remove` and `Revoke` remove capabilities and thus cannot connect two disconnected entities. Moreover, if performed on a sane state, `Create` cannot connect two existing entities: it introduces a new non-overlapping entity and connects that to an existing one. The `Grant` operation, on the other hand has the potential to connect two existing entities, but only under restricted conditions: the grant operation can connect two entities only if they were transitively connected in the state before. Thus, for two existing entities we proved:

Lemma 1. *If two entities of state* `s` *are connected after an execution step, they must have been transitively connected before. Formally:*

$$[\![\texttt{is\_entity}\ s\ e_x;\ \texttt{is\_entity}\ s\ e_y;\ \texttt{step}\ \texttt{cmd}\ s \vdash e_x \leftrightarrow e_y]\!] \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$$

Our plan is to first lift Lemma 1 to the transitive and reflexive closure, such that $\texttt{step}\ \texttt{cmd}\ s \vdash e_x \leftrightarrow^* e_y \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$, for any two existing entities e_x and e_y, by induction over the reflexive transitive closure. Although we are considering the `connected` relationship between existing entities, the proof obligation in the induction step is more general in that it requires us to consider entities that might have been introduced by the current command. It turns out that Lemma 1 is not strong enough to get through the induction step, because it requires both entities to exist in the pre-state. Hence, we break the proof into two parts: we treat `Create` separately from all other commands which we call *transporters*. For transporters, we proved:

Lemma 2. *Transporters preserve* $\texttt{connected}^*$ *in sane states:*

$$[\![\texttt{step}\ \texttt{cmd}\ s \vdash e_x \leftrightarrow^* e_y;\ \texttt{sane}\ s;\ \forall e\ c_1\ c_2.\ \texttt{cmd} \neq \texttt{Create}\ e\ c_1\ c_2]\!] \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$$

For `Create` we proved:

Lemma 3. *Given entities* e_x *and* e_z *in the state after* $\texttt{Create}\ e\ c_1\ c_2$*, given that* e_x *exists in the pre-state* `s`*, and given that* `sane s`*, we know* $s \vdash e_x \leftrightarrow^* e$ *if* e_z *is the entity just created, or* $s \vdash e_x \leftrightarrow^* e_z$ *otherwise. Formally:*

$$[\![\texttt{step}\ (\texttt{Create}\ e\ c_1\ c_2)\ s \vdash e_x \leftrightarrow^* e_z;\ \texttt{is\_entity}\ s\ e_x;\ \texttt{sane}\ s]\!] \Longrightarrow \texttt{if}\ e_z = \texttt{next\_id}\ s\ \texttt{then}\ s \vdash e_x \leftrightarrow^* e\ \texttt{else}\ s \vdash e_x \leftrightarrow^* e_z$$

Then, by induction over the command sequence, together with Lemma 2 and Lemma 3 we conclude:

Theorem 1. *If two entities in a sane state* `s` *are transitively connected after execution, they already have been transitively connected in* `s`*:*

$$[\![\texttt{sane s; is\_entity s } e_x\texttt{; is\_entity s } e_y\texttt{; execute cmds s} \vdash e_x \leftrightarrow^* e_y]\!] \Longrightarrow \texttt{s} \vdash e_x \leftrightarrow^* e_y$$

By computing the symmetric, reflexive and transitive closure over `leak` on state s_1—for which there are a number of well known efficient algorithms (for example [16])—we can predict capability leakages that might happen in future states. However, this closure is an approximation because of the symmetry assumption. Without it, the property is not invariant over the grant and create operations. With this assumption, we might claim that a given subsystem can gain more authority in the future than what it in fact can if there is no transitive self-referential grant capability in the system or if there is no create capability in the system. Although it is possible to build such systems in seL4, and for small, static systems this might even occur in practise, these are very simple to analyse and it is unlikely that the approximation will lead to false alarms. For the majority of systems the invariant and therefore the prediction is precise.

4.2 Subsystems

Formally, we identify a subsystem by any of its entities e_s and define it as the set of entities in the symmetric, reflexive, transitive closure of `leak`:

$$\texttt{subSys :: state} \Rightarrow \texttt{entity\_id} \Rightarrow \texttt{entity\_id set}$$
$$\texttt{subSys s } e_s \equiv \{e_i \mid \texttt{s} \vdash e_i \leftrightarrow^* e_s\}$$

We obtain the entities in a subsystem, using the `subSys` function, specifying the current system state and one of the entities in that subsystem. For instance, in our example the entities of subsystem ss_1 in s_1 are `subSys` s_1 `1 = {1, 2}` (see Fig. 2).

We aim to show that some subsystem, say ss_1, cannot increase its authority over entity e_i. To formally phrase this statement, we introduce two more concepts: the `subSysCaps` function and the *dominates* (`:>`) operator.

$$\texttt{subSysCaps s x} \equiv \bigcup \texttt{caps\_of s `subSys s x}$$
$$\texttt{c :> S} \equiv \forall \texttt{c'} \in \texttt{S. entity c' = entity c} \longrightarrow \texttt{rights c'} \subseteq \texttt{rights c}$$

The `subSysCaps` function initially finds the set of entities in the subsystem, and then returns the union of all capabilities possessed by the entities in that subsystem. A capability `c` dominates a capability set `S` (`c :> S`) if `S` provides at most as much authority as capability `c` over the entity `c` points to.
For isolation to hold we would like to show:

$$\forall \texttt{cmds. c :> subSysCaps (execute cmds } s_1\texttt{)} e_1$$

where `c` is the authority currently possessed by the subsystem over some entity. By using `:>`, we express that authority currently possessed can never grow, as opposed to giving a particular fixed value or restricting ourselves to a particular access right. For isolation to hold we need to show that the above property is true for all states derived from s_1. For a single step of execution we proved:

Lemma 4. *Single execution steps do not increase subsystem authority:*

$$[\![\texttt{sane s; is\_entity s } e_1\texttt{; is\_entity s (entity c); c :> subSysCaps s } e_1]\!] \Longrightarrow \texttt{c :> subSysCaps (step cmd s) } e_1$$

Proof. We begin by noting that no entity that is transitively connected to e_1 possesses a capability with more authority than `c`, formally `c :> subSysCaps s` e_s $= \forall e_x.$ `s` $\vdash e_x \leftrightarrow^* e_s \longrightarrow$ `c :> caps_of s` e_x. Moreover, given that the entity pointed to by `c` already exists and the state is `sane`, the only possibility of obtaining a more authorised capability is by one of the entities within the subsystem receiving a capability via a grant operation. However, for such a grant operation to be `legal` we see that there should be an entity that can `leak` to the subsystem, and hence is `connected` to the entity that received the capability and possesses a more authorised capability than `c`. This is a contradiction to the assumptions.

This leads us to the final isolation theorem:

Theorem 2 (Isolation of authority). *Given a sane state* `s`*, a non-empty subsystem* e_s *in* `s`*, and a capability* `c` *with a target identity* `e` *in* `s`*, if the authority of the subsystem does not exceed* `c` *in* `s`*, then it will not exceed* `c` *in any future state of the system.*

$$[\![\texttt{sane s; is\_entity s } e_s\texttt{; is\_entity s (entity c); c :> subSysCaps s } e_s]\!] \Longrightarrow \texttt{c :> subSysCaps (execute cmds s) } e_s$$

This concludes our isolation proof. The authority that a subsystem collectively has over another entity cannot grow beyond what is conferred by the resource manager initially. Going back to our initial example in Fig. 2, this means that no entity in subsystem ss_1 will ever gain more authority than `{W}` over entity `3`. Moreover, none of these entities will ever gain any authority over entity `i`.

The statement of Theorem 2 has assumptions about entities existing in state `s` before execution. Since the authority bound is over whole subsystems and not particular entities, the theorem also covers entities that are created during execution and do not exist in `s` yet, as long as they belong to a subsystem that is spanned by an existing entity. Since the `Create` operation always connects the new entity to an existing one, we are covered.

What about new subsystems as opposed to new entities, though? The only way to create new subsystems is to remove grant arcs between entities such that two parts of an existing subsystem become disconnected. In the state `s'` before that removal, Theorem 2 ensures that none of the two candidates have violated their authority bounds. In the state directly after removal, authority has only been removed from the system, so neither subsystem has gained authority. Now we can apply Theorem 2 again. The problem is merely naming the intermediate new subsystems and their parentage.

The main conclusion including new subsystems is: no subsystem can gain more authority than what it already possesses to your resources unless you created it yourself, in which case it cannot exceed what you gave it.

4.3 Implementing Subsystems

We now show how the isolated subsystems described above are bootstrapped and implemented in seL4. Recall that after system startup, the initial state s_0 contains only the resource manager with full access rights to itself and with the authority over all the physical memory that is not used by the kernel.

For each of the subsystems the resource manager creates a subsystem resource manager who is responsible for bootstrapping the rest of that particular subsystem. This delegation scheme stems from a major application domain of seL4: running para-virtualised operating systems in each subsystem.

Coupled with the resource manager is a specification language that is used by the developer to specify which subsystems should be created together with what authority they should possess and how much physical memory should be committed to each subsystem. Given below is such a specification:

```
"ss1" { text {1024 to 4096}; data {4096 to 5120 };
        resource {64};       comm {this → "ss2"}; };
"ss2" { text {5120 to 6144}; data {6144 to 10240 };
        resource {64};       comm {this → "ss1"}; };
```

This system would constitute two subsystems: *ss1* and *ss2*, each with authority to send information to the other—specified by *comm {this → ssX}*, and access to 64KB physical, untyped memory (*resource {64}*). The keywords *text* and *data* specify where to find the text and the data segments of each subsystem resource manager respectively.

For *ss1* and *ss2* to be authority isolated, the resource manager should guarantee $\neg\ s \vdash ss1 \leftrightarrow^* ss2$. This is achieved by construction—there is no language construct to specify grant authority between subsystems. Note that the subsystem managers are still free to provide grant authority within the subsystems. We merely exclude the possibility for authority to leak between partitions. A small compiler then translates this specification to a sequence of kernel operations for execution by the resource manager at boot time.

Part (a) of Fig. 5 shows the initial state of our resource manager. The configuration after creating and populating each entity in accordance with the above specification is given in part (b) of Fig. 5. However, in this state, both entity 1 and 2 are still connected through the resource manager, hence formally inhabit

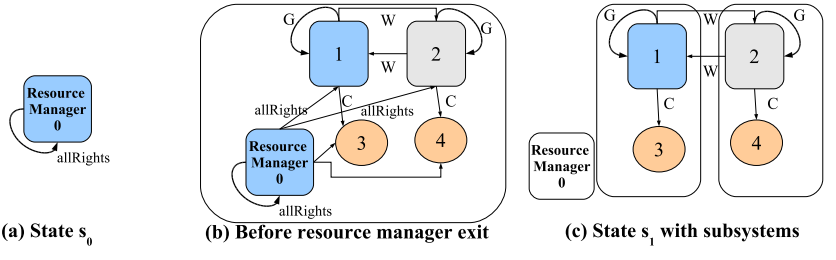


Fig. 5. Subsystem Configuration

```
[Create 0 (allCap 0) (allCap 0), Create 0 (allCap 0) (allCap 0),
 Create 0 (allCap 0) (allCap 0), Create 0 (allCap 0) (allCap 0),
 Grant 0 (allCap 1) (allCap 1) {G}, Grant 0 (allCap 1) (allCap 2) {W},
 Grant 0 (allCap 1) (allCap 3) {C}, Grant 0 (allCap 2) (allCap 2) {G},
 Grant 0 (allCap 2) (allCap 1) {W}, Grant 0 (allCap 2) (allCap 4) {C},
 Remove 0 (allCap 0) (allCap 1), Remove 0 (allCap 0) (allCap 2),
 Remove 0 (allCap 0) (allCap 3), Remove 0 (allCap 0) (allCap 4)]
```

Fig. 6. Sequence of kernel operations for bootstrapping

the same single subsystem. The final task of the resource manager therefore is to break these grant arcs—once bootstrapped, the resource manager removes its own capabilities to entities and exits, thereby arriving at part (c) of Fig. 5. We call this state s_1, the initial state of the untrusted user mode system.

One possible sequence of commands the resource manager (entity 0) can execute to produce s_1 is given in Fig. 6 (for convenience shown left to right).

The closed term for the initial state is s_1 = (|heap = [1 ↦ {gCap 1, wCap 2, cCap 3}, 2 ↦ {gCap 2, wCap 1, cCap 4}], next_id = 5|), where gCap e, wCap e, and cCap e stand for a capability to entity e with G, W, and C rights respectively. The term noCap e in the lemma below stands for a capability to entity e without any access rights.

Strictly speaking, there are 5 subsystems in s_1, each with one entity, but only the entities 1 and 2 contain capabilities. They constitute the main two subsystems. We can now, for example show the following.

Lemma 5. *For no sequence of commands can the subsystem 1 gain authority over entity 4 which stands for the physical memory resources of subsystem 2.*
∀ cmds. noCap 4 :> subSysCaps (execute cmds s_1) 1

5 Related Work

Harrison et al. [10] first formulated access control analysis in a model known as *HRU* and focused on the ability of a subject to obtain a particular authority over another in some future state. Our analysis differs from HRU in that we focus on the collective authority possessed by a set of entities.

As mentioned earlier, the take-grant (*TG*) model [13] is closely related to the seL4 protection model. The original analysis of de jure access rights on the TG model [2,13] already uses the same approximation to model the exposure of access rights: the transitive, symmetric closure on the given initial graph. Our model is different to the classic TG model in that it is aimed at reasoning over the distribution and control of physical resources like memory. We do not use the take-rule in seL4. This has the advantage of giving each subject control over the distribution of its authority at the source. Additionally, to better model the accounting for physical memory, we have added a more complex create rule. Our proof shows that the desirable properties of TG still hold and can be generalised to a statement on full subsystems.

Snyder [20] and later Bishop [1] enhanced the TG model by introducing de facto rules—rules that derive feasible information flow paths given the capability distribution. They used the term *island* to denote a maximum take-grant connected subgraph which is similar to our concept of subsystem. The analysis we presented can be extended easily to de facto rights.

Shapiro [18] applied the *diminish-take* model—another variant of TG to capture the operational semantics of the *EROS* system.

All formalisations and proofs mentioned in the works above are pen-and-paper only, mostly using graph diagram notation. Our model and proof are formalised and machine-checked in Isabelle/HOL, making the argument fully precise. While we can affirm the previous general results for our model, we did find that graph diagrams can often be deceptively simple, glossing over subtle side conditions that the theorem prover enables us to track precisely. Examples of such side conditions are: new nodes added to the graph cannot overlap with the existing ones (a condition explicitly tested in the kernel), and the graph cannot have dangling arcs. The precise statement of our theorem also makes clear that at each time it covers only existing entities. Although it can be applied inductively for these situations, the issue does not become apparent in earlier formalisations.

Rushby [17] goes further in providing a formulation of isolation that is called non-interference. In this paper we only consider access control and propagation of authority. The non-interference property is stronger: It covers full information flow and timing channels. While such an analysis on the seL4 design would certainly be worthwhile, we do not believe that a strong information flow property can be established by the implementation on current mainstream hardware. Our access control model on the other hand can be.

6 Conclusions

In this paper, we have presented a machine-checked, high-level security analysis of the seL4 microkernel. We have formalised an access control model of seL4 in the interactive theorem prover Isabelle/HOL. The formalisation is inspired by the classical take-grant model, but without the take rule and with a more complex, realistic, create rule for achieving precise control of memory allocation. Our formalisation makes the graph diagram notation that is used for this type of analysis in the literature fully precise.

We have shown, in Isabelle/HOL, that seL4 mechanisms are sufficient to enforce mandatory isolation between subsystems and that collective authority of subsystems does not increase. Through an example we have shown that the restrictions required for isolation are pragmatic, and we have successfully implemented a resource manager capable of bootstrapping a paravirtualised Linux kernel [11] on seL4.

Since all memory is controlled directly by capabilities in seL4, isolated subsystems are fully spatially separated from one another. The model is general enough to also allow for explicit information flow across subsystem boundaries via read/write operations. Our main theorem shows that subsystems can neither

exceed their authority over physical memory nor their authority over communication channels to other subsystems.

Future work includes the de facto rights analysis which, as mentioned above, should be easy to add. More importantly future work also includes a formal refinement between the model presented here and our work on the binary compatible seL4 API model [5]. The aim is to make our security analysis apply directly to the full C and assembler implementation of seL4 on the ARM11 platform. The model presented here is a slightly simplified version of the one that we intend to use for refinement. The main difference in the refinement model is the use of non-determinism to indicate potential failures like running out of memory on object creation.

References

1. Bishop, M.: Conspiracy and information flow in the take-grant protection model. Journal of Computer Security 4(4), 331–360 (1996)
2. Bishop, M., Snyder, L.: The transfer of information and authority in a protection system. In: 7th SOSP, pp. 45–54. ACM Press, New York (1979)
3. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Munoz, C., Ait, O. (eds.) TPHOLs 2008. LNCS 5170 (to appear, 2008)
4. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. CACM 9, 143–155 (1966)
5. Derrin, P., Elphinstone, K., Klein, G., Cock, D., Chakravarty, M.M.T.: Running the manual: An approach to high-assurance microkernel development. In: ACM SIGPLAN Haskell WS, Portland, OR, USA (September 2006)
6. Elkaduwe, D., Derrin, P., Elphinstone, K.: A memory allocation model for an embedded microkernel. In: Proc. 1st MicroKernels for Embedded Systems, Sydney, Australia, pp. 28–34. NICTA (January 2007)
7. Elphinstone, K., Klein, G., Derrin, P., Roscoe, T., Heiser, G.: Towards a practical, verified kernel. In: 11th HotOS, San Diego, CA, USA (May 2007)
8. Elphinstone, K., Klein, G., Kolanski, R.: Formalising a high-performance microkernel. In: Leino, R. (ed.) Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 2006), Microsoft Research Technical Report MSR-TR-2006-117, pp. 1–7, Seattle, USA (August 2006)
9. Hardy, N.: KeyKOS architecture. Operat. Syst. Rev. 19(4), 8–25 (1985)
10. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in operating systems. In: CACM, pp. 561–471 (1976)
11. Leslie, B., van Schaik, C., Heiser, G.: Wombat: A portable user-mode Linux for embedded systems. In: 6th Linux.Conf.Au, Canberra (April 2005)
12. Liedtke, J.: On μ-kernel construction. In: 15th SOSP, Copper Mountain, CO, USA, pp. 237–250 (December 1995)
13. Lipton, R.J., Snyder, L.: A linear time algorithm for deciding subject security. J. ACM 24(3), 455–464 (1977)
14. National ICT Australia. seL4 Reference Manual (2006), `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`
15. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

16. Nuutila, E.: Efficient transitive closure computation in large digraphs. PhD thesis, Helsinki University of Technology (June 1995)
17. Rushby, J.: Noninterference, transitivity, and channel-control security policies. Technical report (December 1992), http://www.csl.sri.com/papers/csl-92-2/
18. Shapiro, J.S.: The practical application of a decidable access model. Technical Report SRL-2003-04, SRL, Baltimore, MD 21218 (November 2003)
19. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: A fast capability system. In: 17th SOSP, Charleston, SC, USA, pp. 170–185 (December 1999)
20. Snyder, L.: Theft and conspiracy in the Take-Grant protection model. Journal of Computer and System Sciences 23(3), 333–347 (1981)

Verification of the Deutsch-Schorr-Waite Marking Algorithm with Modal Logic

Yoshifumi Yuasa[1,4], Yoshinori Tanabe[2,4], Toshifusa Sekizawa[3,4], and Koichi Takahashi[4]

[1] Graduate School of Information Science and Engineering, Tokyo Institute of Technology
[2] Graduate School of Information Science and Technology, University of Tokyo
[3] Graduate School of Information Science and Technology, Osaka University
[4] Research Center for Verification and Semantics (CVS), National Institute of Advanced Industrial Science and Technology (AIST)

Abstract. We have proposed an abstraction technique that uses the formulas of variants of the modal μ-calculus as a method for analyzing pointer manipulating programs. In this paper, the method is applied to verify the correctness of the Deutsch-Schorr-Waite marking algorithm, which is regarded as a benchmark of such analysis. Both the partial correctness and the termination are discussed. For the former, we built a system on top of the proof assistant Agda, with which the user constructs Hoare-style proofs. The system is an optimum combination of automatic and interactive approaches. While a decision procedure for a variant of modal μ-calculus, which is available through the Agda plug-in interface, enables the user to construct concise proofs, the run time is much shorter than for automatic approaches.

1 Introduction

The Deutsch-Schorr-Waite (DSW) algorithm is a marking algorithm for garbage collection. An ordinary marking algorithm based on depth first search uses a stack to remember the node to return to after completing the marking from the current node. The DSW algorithm does not use a stack. Instead, it temporarily alters the pointers of the current node to remember the node to return to. When the marking from the current node is completed, it returns to the remembered node and restores the pointers.

The correctness of the DSW algorithm is not trivial, and it has been regarded as a benchmark of verification methods for pointer manipulating programs. In [9,10], we proposed a method to analyze pointer manipulating programs using modal logics. In this research, we apply the method to verify the correctness of the DSW algorithm.

In our approach, a heap is regarded as a variant of the Kripke structure, which we call a pointer structure. We regard each cell in the heap as a state of the Kripke structure, a "points-to" relation by a pointer-type field of the cell as a labeled transition relation, a boolean-type field of the cell as a predicate

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 115–129, 2008.

symbol (an ordinary atomic formula), and a pointer-type program variable as a nominal (an atomic formula that is satisfied by only one state). We define a variant of modal μ-calculus to describe heap statuses. Due to the fixed-point operators, we can express the various properties of a heap. Moreover, the language has two features that we utilize in verification. First, we can express the weakest preconditions and the strongest postconditions of a formula with respect to the pointer operations by a formula of the logic [10]. Second, the satisfiability problem of the logic is decidable. We have implemented an efficient procedure to solve the problem for the alternation-free fragments of the logic [11], which have a sufficient expressive power for our aim. Combining these two features, we can judge whether a Hoare triple "$A\,\{cs\}\,B$" holds or not, where A and B are formulas of the logic and cs is a basic block of a program.

We built a system to construct Hoare-style proofs on top of the proof assistant Agda [1]. A validity checker of the abovementioned Hoare triples is implemented as a command-line program that the user calls through the plug-in mechanism of Agda. Using the system, we verify the partial correctness of the DSW algorithm.

Our approach is in a sense a middle ground between the automatic and interactive methods. The user of the system manually constructs a proof in Hoare logic, with the help of the validity checker.

The burden on the user is lighter than writing an entire proof since some part of the proof is automatically carried out by the validity checker. Moreover, although invariants of while clauses need to be manually constructed, the system helps to find them: based on the intuition, the user can enumerate statuses of the heap when the while clause is executed and define the transitions between them, as we illustrate in Section 5.2. The correctness of the transitions is verified by the validity checker. Then, the disjunction of the formulas that define the statuses is a suitable invariant for the while clause. Note that even in the automated system, ingredients of the invariants need to be given to the system.

The running time of our system is much shorter than the automatic systems. This is because while the automatic systems need to construct (typically) big transition systems, we calculate only what the user explicitly requests.

As a result, the proof we construct for the partial correctness of the DSW algorithm consists of a few hundred lines in the Agda language and the total running time of the validity checker is less than three minutes.

For a termination proof of the DSW algorithm in the "heaps as Kripke structures" framework, we define three formulas that induce ranking functions. The correctness is reduced to the validity of some formulas constructed from them.

We briefly review related work. The DSW algorithm was presented in its original form in [8]. One of the earliest studies to give a formal proof that can be checked by computer is [2] by Bornat, in which the proof assistant Jape was used. Mehta and Nipkow used the higher-order system Isabelle/HOL in [6]. Yang wrote a formal proof in the logic called Bunched Implications [12], which becomes the basis of separation logic.

In the abovementioned studies, proofs were constructed manually and then checked by proof assistants on computer. Hubert and Marché [4] used a tool

called CADUCEUS, which handles C source code directly, and invariants can be embedded into the code as comments in a special form. The tool verifies that they are really invariants.

Loginov *et al.* automated the verification [5]. They use the tool TVLA [7], which performs abstract interpretation of pointer manipulating programs. Since the default predicates in TVLA are insufficient to verify the properties of the algorithm, they added predicates for this verification. TVLA completes the verification in several hours, verifying both the partial correctness and termination; however, the input data is restricted to the shape of a tree.

The remainder of the paper is organized as follows: In Section 2, we define a tiny programming language, which we use throughout the paper, and then show how to describe the DSW marking algorithm in that language. The section includes an explanation of three important sub-processes push, swing and pop of the DSW algorithm. Section 3 introduces the concept of "heaps as Kripke structures," on which our verification framework is based. Two key theorems and a validity checker of Hoare triples based on them are described. In Section 4, we briefly explain the proof assistant Agda, the front-end of our system, and describe a library we provide on the top of Agda to help users reasoning in the Hoare logic. In Section 5, we show the correctness of the DSW algorithm. A proof of the partial correctness built with our system is described in detail, and termination is also discussed. Section 6 concludes the paper.

2 Deutsch-Schorr-Waite Marking Algorithm

2.1 Programming Language

Through this paper, we describe pointer algorithms in a tiny procedural programming language. We assume that cells in the heap we manipulate are of the same type, *i.e.*, they all have pointer-type fields indexed by F and bit-fields (boolean-type fields) indexed by B for fixed finite sets F and B of letters. We also fix another finite set V of program variables. All program variables are of pointer-type, *i.e.*, they either hold cells or have the value "null". We introduce a program constant "`Null`" holding the null, and denote $\mathsf{V} \cup \{\texttt{Null}\}$ by V^*.

Here, E_0 and C_0 are the sets of *atomic expressions* and *basic operations* on cells, as listed below.

$$\begin{aligned}\mathsf{E}_0 &::= \mathsf{V} \texttt{ == Null} \mid \mathsf{V} \texttt{ == } \mathsf{V} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ == True} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ == False}\\ \mathsf{C}_0 &::= \mathsf{V} \texttt{ := Null} \mid \mathsf{V} \texttt{ := } \mathsf{V} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ := True} \mid \mathsf{V}\texttt{.}\mathsf{B} \texttt{ := False} \mid\\ &\qquad \mathsf{V} \texttt{ := } \mathsf{V}\texttt{.}\mathsf{F} \mid \mathsf{V}\texttt{.}\mathsf{F} \texttt{ := } \mathsf{V}\end{aligned}$$

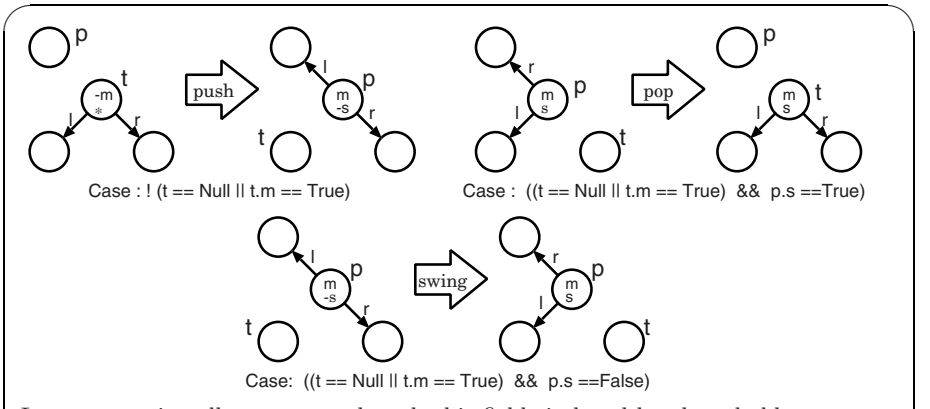
Then we define the *boolean expressions* E, *clauses* C, and *programs* P in the language, where the symbol ϵ represents an empty sequence.

$$\begin{aligned}\mathsf{E} &::= \mathsf{E}_0 \mid \texttt{(!}\mathsf{E}\texttt{)} \mid \texttt{(}\mathsf{E} \texttt{ \&\& } \mathsf{E}\texttt{)} \mid \texttt{(}\mathsf{E} \texttt{ || } \mathsf{E}\texttt{)}\\ \mathsf{C} &::= \mathsf{C}_0 \mid \texttt{if } \mathsf{E} \texttt{ then \{}\mathsf{P}\texttt{\} else \{}\mathsf{P}\texttt{\}} \mid \texttt{while } \mathsf{E} \texttt{ do \{}\mathsf{P}\texttt{\}}\\ \mathsf{P} &::= \epsilon \mid \mathsf{C}\texttt{;}\mathsf{P}\end{aligned}$$

In addition, any white space and comments bracketed by '`/*`' and '`*/`' can be inserted arbitrarily in the program in the usual manner.

```
t := root; p := Null;                                  /* init  */
while (!(p == Null) || !(t == Null || t.m == True)) do {
  if (!(t == Null || t.m == True)) then {              /* push  */
     x := p; p := t  ; t := t.l; p.l := x;
     p.s := False; p.m := True;
   } else { if (p.s == False) then {                   /* swing */
     x := t; t := p.r; y := p.l; p.r := y; p.l := x;
     p.s := True ;
   } else {                                            /* pop   */
     x := t; t := p  ; p := p.r; t.r := x;
   }; };
 };
```

Fig. 1. The DSW algorithm



Letters m, s in cells represent that the bit-fields indexed by them hold true, $-m$, $-s$ represents they hold false, and "$*$" represents we don't concern the field.

Fig. 2. Operations with the three in-line procedures in the while loop

2.2 DSW Algorithm in the Tiny Language

Figure 1 presents the DSW algorithm described in our language. We will restrict ourselves to deal only with the case where a cell has two pointer-indices *l*eft and *r*ight, as in [4] and [5]. The bit-indices are *m*ark and *s*wung. The mark-fields of cells reachable from the root-cell are set to be true during the process and the swung-fields indicate whether the cell has been processed with the swing procedure described below.

At a point in execution time, we call the cell held by the variable p a *current cell*, and the cell held by t a *next cell*. We start with setting the current cell to null and the next cell as the root-cell in the heap, and then go into the while loop, in which one of the three in-line procedures is operated depending on the conditions of the current and next cells (see Fig.2). We exit the while loop if the current cell is null and the next cell is either null or marked.

In the following argument, the three in-line procedures in the while loop are termed *push*, *swing*, and *pop*, as in [4]. Push and pop correspond to the procedures used by typical depth-first marking algorithms with a stack. In the DSW algorithm, we do not use a stack to remember the cell to return to, but use the pointer a cell having been pointed by which we are marking from. The swing procedure exchanges the pointer from left to right, when we have completed marking from the left cell.

3 Heaps as Kripke Structures

3.1 Pointer Structures

We represent a heap status by a directed graph structure with additional information, regarding the cells as the nodes and the pointers as the edges. Formally, we consider a quintuple $(S, nil, \{p_f\}_{f\in\mathsf{F}}, \rho_0, \rho_1)$ consisting of:

- A finite set S of cells and $nil \in S$ for the destination of any null pointers,
- A set of points-to functions $p_f : S \to S$ indexed by F assuming each p_f maps nil to nil,
- A function $\rho_0 : \mathsf{B} \to \mathcal{P}S$ mapping a bit-index to the set of cells holding "true" in the bit-field, and
- A function $\rho_1 : \mathsf{V}^* \to S$ mapping a variable to the cell held by it (or to nil) and the constant `Null` to nil.

We call such a quintuple a *pointer structure*. A pointer structure P induces a Kripke structure $K(P) = (S, \{R_f\}_{f\in\mathsf{F}}, \rho)$ on the propositional variables $\mathsf{B} \cup \mathsf{V}^*$ with R_f's and ρ defined as:

$$s\, R_f\, t \Leftrightarrow p_f\, s = t, \qquad \rho(x) = \begin{cases} \rho_0(x) & (\text{if } x \in \mathsf{B}\ ) \\ \{\rho_1(x)\} & (\text{if } x \in \mathsf{V}^*). \end{cases}$$

We now introduce a formal language to describe heap statuses. This paper defines the language as a variant of *modal μ-calculus*, although the definition below works for various modal languages. The reader is assumed aware of the language and the semantics of modal μ-calculus. We use the basic notions and terms about it following [13]. In the following argument, we call a formula in the modal μ-calculus a *modal formula*.

We call a formula in the form "$@v\,\varphi$" a *basic p-formula*, for a program variable v and a modal formula φ with modalities from F and atomic propositions from $\mathsf{B} \cup \mathsf{V}^*$. Its satisfaction by a pointer structure P is defined as:

$$P \models @v\,\varphi \Leftrightarrow K(P), \rho_1(v) \models \varphi,$$

where the '$\models$' on the right-hand side represents the satisfaction for the modal μ-calculus. In other words, a pointer structure satisfies $@v\,\varphi$, if the induced Kripke structure satisfies φ at $\rho_1(v)$. We call boolean combinations of basic p-formulas *p-formulas*; satisfaction is extended in the usual way.

A basic p-formula $@v\,\varphi$ is called to be *alternation-free* if φ is alternation-free, *i.e.*, of alternation depth less than 2. Note that any CTL formula is equivalent to an alternation-free modal formula. A p-formula is called to be alternation-free if its components are all alternation-free.

3.2 Examples of P-Formulas

Figure 3 shows some examples of p-formulas; the pointer structure satisfies the three p-formulas. There are four schemata defining modal formulas $\boldsymbol{\alpha}\,\varphi$, $\boldsymbol{\beta}\,\varphi$, $\boldsymbol{\gamma}\,\varphi$, and $\boldsymbol{\delta}\,\varphi$ for a modal formula φ, which are taken from the argument to prove the partial correctness of the DSW algorithm in Section 5. Note that they are alternation-free, if φ is alternation-free.

The pointer structure on the right satisfies the following p-formulas:

$$@x\,(\boldsymbol{\beta}\,w), \quad @y\,(\boldsymbol{\gamma}\,z), \quad @x\,(\boldsymbol{\delta}\,z),$$

where :

$$\begin{aligned}
\boldsymbol{\alpha}\,\varphi &= (s \wedge \Diamond_r\varphi) \vee (\neg s \wedge \Diamond_l\varphi)\\
\boldsymbol{\beta}\,\varphi &= \mu X.(\varphi \vee (m \wedge \boldsymbol{\alpha}\,X))\\
\boldsymbol{\gamma}\,\varphi &= \mu X.(\neg m \wedge (\varphi \vee \Diamond_l X \vee \Diamond_r X))\\
\boldsymbol{\delta}\,\varphi &= \boldsymbol{\beta}(\Diamond_r(\boldsymbol{\gamma}\,\varphi))
\end{aligned}$$

$\mathsf{F} = \{l\mathrm{eft}, r\mathrm{ight}\}$, $\mathsf{B} = \{m\mathrm{ark}, s\mathrm{wung}\}$,
$\mathsf{V} = \{x, y, z, w\}$

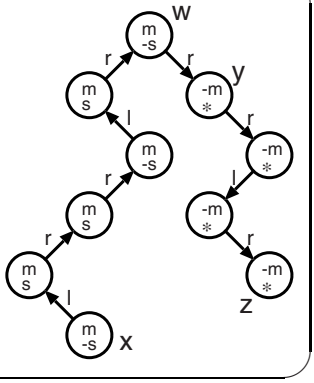


Fig. 3. Examples of p-formulas

The formula $\boldsymbol{\alpha}\,\varphi$ says that the formula φ holds at either the left successor or the right successor cell depending on the swung-field. Then, the formula $\boldsymbol{\beta}\varphi$ represents the reachability to a cell with the property φ, through a path on which all cells have the value "true" in their mark-fields and all pointers to succeeding cells are selected in the same way as in $\boldsymbol{\alpha}$. The formula $\boldsymbol{\gamma}\,\varphi$ represents a more liberal reachability to a cell with φ. One can go through a path selecting any desired left successors and right successors, as long as they have the value "false" in their mark-fields. The schema $\boldsymbol{\delta}$ is a combination of $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$.

3.3 Strongest Post-conditions and Satisfiability Problem

The reason why we choose the language in this paper is that it has two desired properties, which are described in the following key theorems of our framework, where we call a program a *basic block* if it consists of basic operations, and denote by "$cs\,P$" the pointer structure obtained from a pointer structure P by operating a program cs.

Theorem 1. *Let cs be any basic block. There exists a procedure spc which calculates the strongest post condition of a p-formula with respect to cs, i.e., $cs\,P \models spc(A, cs)$ if and only if $P \models A$ for any pointer structure P and any p-formula A. Particularly, $spc(A, cs)$ is an alternation-free formulas, if A is alternation-free.*

Theorem 2. *There exists a procedure "sat" defined on the alternation-free p-formulas, which determines whether a given p-formula can be satisfied, i.e., sat A is true if and only if there is a pointer structure that satisfies A.*

See [10] and [11] for the proofs of these theorem. The last part of Theorem 1 is not mentioned in [10], but it is clear from the proof shown there.

As suggested by Theorem 2, the expressive power of p-formulas is weaker than languages used in frameworks such as [5] and [12]. Still, all the necessary predicates to prove the correctness of the DSW algorithm can be expressed as we will see in Section 5. Also, note that the procedure mentioned in the theorem judges whether a pointer structure, instead of Kripke structure, exists or not. Although the satisfaction relation of a pointer structure is defined by regarding it as a Kripke structure, there are Kripke structures that cannot be seen as a pointer structure. The procedure is aware of the difference for precise analysis.

Then, we define:

$$\begin{aligned} pftrans(A, cs, B) &= \quad sat\,(spc(A, cs) \wedge \quad B), \\ pfvalid(A, cs, B) &= \neg sat\,(spc(A, cs) \wedge \neg B). \end{aligned}$$

The former determines if the operation cs changes *some* pointer structure with property A to a structure with property B, on the other hand, the latter determines if the operation cs changes *all* pointer structures with A to structures with B. A Hoare-triple "$A\,\{cs\}\,B$" is clearly valid if "$pfvalid(A, cs, B)$" is true.

The system presented in the paper implements *pfvalid* in Java with BDD library, and uses it to help user's Hoare-style reasoning with a proof assistant.

Based on the concept in this section (but using 2-way CTL instead of the μ-calculus), we developed another system called Modal Logic Abstraction Tool (MLAT), which supports automatic reasoning on pointer algorithms. It implements *pftrans* to create a model by predicate abstraction. See Sekizawa *et al.* [9] for more information about MLAT.

4 Hoare Logic on the Proof Assistant

4.1 Agda, a Proof Assistant

We use an interactive proof assistant *Agda* for the front-end of our tool. We briefly describe the system here. See [1] for further information.

Agda is developed in Chalmers University of Technology. Its input language, called the *Agda language*, is based on Martin-Löf's constructive type theory. The concrete syntax is similar to those of functional programming languages, especially of Haskell. A logic is encoded into Agda language by Curry-Howard

isomorphism. Formulas are encoded as types. For example, a conjunction of formulas is encoded as a Cartesian product of types. Users prove theorems by constructing terms, called *proof terms*, inhabiting the theorems encoded as types. For example, a proof term of the distributive law of propositional logic is defined as follows, provided `cnjIntr`, `cnjElimL`, ... are encoded inference rules of the natural deduction system. (`cnjIntr` is a pairing function as which we encode the introduction rule for conjunction, ...*etc.*)

```
dstr (A,B,C :: Prop) :: A && (B || C) -> (A && B) || (A && C)
     = \(p :: A && (B || C)) ->
       let q ::  A       = cnjElimL p ;
           r :: (B || C) = cnjElimR p ;
        in dsjElim r (\(x :: B) -> dsjIntrL (cnjIntr q x))
                     (\(y :: C) -> dsjIntrR (cnjIntr q y))
```

Agda checks types of user's inputs as above. A proof is correct, if it is well-typed.

Users can omit some arguments of a function, called *hidden arguments*, if they are inferred from the rest of arguments. For example, the first two arguments `A` and `(B || C)` of cnjElimL in the proof above are omitted. In fact, arguments of a function are hidden by default. Users declare arguments not hidden by marking them with "`!`" when they define the function. (See Fig.7 for example.)

Agda is equipped with a general plug-in mechanism, through which we execute any external prover that has a command line interface. To invoke an external prover, we put a keyword "external" followed by its plug-in name, on the right-hand side of a defining equation of a proof term. For the example above, we prove the law as:

```
dstr (A,B,C :: Prop) :: A && (B || C) -> (A && B) || (A && C)
     = external "fol"
```

where "fol" is a plug-in name of a prover for first-order logic on the command line. Agda regards the definition to be type correct, if the prover answers positively.

4.2 Hoare Logic Library

We provide a library, consisting of about 5000 lines, for Hoare-style reasoning on our programming language. The following items are encoded into Agda language:

- The programming language (boolean expressions, clauses, and programs),
- Modal formulas, p-formulas, and their sequent-style inference rules,
- Hoare triples and their inference rules.

The encodings are done in the manner of *deep-embedding*; for example, the programming language and the p-formulas are encoded as data types, then, the Hoare triples are encoded as an inductive type-family on these data types. A type constructor HTc makes the Hoare triple of a clause and two p-formulas (pre- and post-conditions). Another type constructor HTp is similar, but its first argument is a program. The inference rules of Hoare logic are shown in

Rules in the Hoare logic:

$$\frac{spc(A,c;) \vdash B}{A\,\{c\}\,B}\text{atmHc} \qquad \frac{A \wedge \bar{s}\,\{cs_0\}\,B \quad A \wedge \neg\bar{s}\,\{cs_1\}\,B}{A\,\{\texttt{if}\ s\ \texttt{then}\ \{cs_0\}\ \texttt{else}\ \{cs_1\}\}\,B}\text{iteHc} \qquad \frac{A \vdash B}{A\,\{\epsilon\}\,B}\text{nilHp}$$

$$\frac{A \vdash C \quad C \wedge \bar{s}\,\{cs\}\,C \quad C \wedge \neg\bar{s} \vdash B}{A\,\{\texttt{while}\ s\ \texttt{do}\ \{cs\}\}\,B}\text{whlHc} \qquad \frac{A\,\{c\}\,C \quad C\,\{cs\}\,B}{A\,\{c;cs\}\,B}\text{cnsHp}$$

Partially automated rules (bold face letters represent finite sets of formulas) :

$$\frac{\bigwedge \boldsymbol{A}_0 \vdash B}{\bigwedge \boldsymbol{A}_1 \vdash B}\text{cnjSup}(\boldsymbol{A}_0 \subseteq \boldsymbol{A}_1) \qquad \frac{A \vdash \bigwedge \boldsymbol{B}_0 \quad A \vdash \bigwedge \boldsymbol{B}_1}{A \vdash \bigwedge \boldsymbol{B}_2}\text{cnjSub}(\boldsymbol{B}_2 \subseteq \boldsymbol{B}_0 \cup \boldsymbol{B}_1)$$

$$\frac{\bigvee \boldsymbol{A}_0 \vdash B \quad \bigvee \boldsymbol{A}_1 \vdash B}{\bigvee \boldsymbol{A}_2 \vdash B}\text{dsjSub}(\boldsymbol{A}_2 \subseteq \boldsymbol{A}_0 \cup \boldsymbol{A}_1) \qquad \frac{A \vdash \bigvee \boldsymbol{B}_0}{A \vdash \bigvee \boldsymbol{B}_1}\text{dsjSup}(\boldsymbol{B}_0 \subseteq \boldsymbol{B}_1)$$

Fig. 4. Some inference rules in the library

Fig.4, where $\bar{s}$ denotes a p-formula with the same meaning as an expression s; for example, by considering the semantics of basic p-formulas, we have :

$$\overline{(x == y)} = @x\,y \quad \text{and} \quad \overline{(x.b == false)} = @x\,\neg b.$$

These rules are encoded as constructors of encoded Hoare triples. For example, the encoded rule for the if-then-else clause is typed as:

```
iteHc (s :: Expr) (cs, ds :: Prog) (!A, B :: PForm)
  :: HTp cs (A && s2f s) B -> HTp cs (A && not (s2f s)) B ->
     HTc (if_then_else s cs ds) A B
```

A user may prove a Hoare triple "$A\,\{cs\}\,B$" by invoking the external prover *pfvalid* through the plug-in mechanism, instead of writing down a proof term, if cs is a basic block and A, B are alternation-free. In particular, by taking the empty sequence for cs, we prove a logical sequent "$A \vdash B$" by the prover.

We also implement, by the technique called *reflection* [3], a simple "internal" prover on finite sets, depending not on any external prover, but on the type checking mechanism of Agda itself. Although internal provers are not so powerful as external provers, there is an advantage of using them: One can define a function with a type involving calls to internal provers, and then execute the provers every time the function is used, which is not possible with external provers. This way, we partially automate some derived rules in the library. See the lower part of Fig.4, where the conditions in parentheses are tested automatically before the rules are applied. These rules help a user out of many boring applications of associative and commutative rules of the boolean operators. A user can also define new automated rules on the top of the library for one's own sake. In this research, we define the following rule used in Sec. 5.2:

$$\frac{\{\ C \wedge A \wedge \bigvee \boldsymbol{B}_i\ \{cs\}\ A \wedge \bigvee \boldsymbol{D}_i\ \}_{i\in I}}{C \wedge A \wedge \bigvee \boldsymbol{B}\ \ \{cs\}\ A \wedge \bigvee \boldsymbol{D}}\text{lem3}(\boldsymbol{B} \subseteq \textstyle\bigcup_i \boldsymbol{B}_i, \bigcup_i \boldsymbol{D} \subseteq \boldsymbol{D})$$

which combines proofs of the relationships between states in an abstract transition model to prove correctness of the loop invariant obtained from the model.

5 Verification of the DSW Algorithm

5.1 Outline of the Verification

We now verify the partial correctness of the DSW marking algorithm, which consists of the following:

1. The points-to relationships between cells are preserved.
2. A cell reachable from the root-cell is marked.
3. A cell not reachable from the root-cell is not marked.

We first prove property 3, the easiest among them, to illustrate our method. Properties 1 and 2 are shown in a similar manner. However, the loop invariant needed to prove them is complicated, while the invariant for property 3 is simple enough for our external prover to check directly. In section 5.2, we present an idea for managing such a situation.

To show property 3, we construct a proof term of "$Pre\,\{dsw\}\,Pos$" encoded to the Agda language, letting:

$$\begin{aligned} Pre &= @a\,(\neg m \wedge \neg \texttt{Null}) \wedge @root\,\boldsymbol{\eta}\,a \\ &\qquad\qquad \text{where } \boldsymbol{\eta}\,\varphi = \neg\mu X.(\varphi \vee \Diamond_l X \vee \Diamond_r X), \\ Pos &= @a\,(\neg m). \end{aligned}$$

The program variable a should be fresh, so as to show property 3 for any cell in the heap. Figure 5 presents the proof, in which we proceed as follows:

i) Show the Hoare triple "$Pre\,\{init\}\,Inv$", which ensures we enter the while loop with the status Inv, a loop invariant,
ii) Check the loop invariant, first for the procedures push, swing, and pop, and then for the loop body as a result,
iii) Show "$\neg C \wedge Inv \vdash Pos$" for the loop condition C, which ensures we exit the loop with the status Pos, and then
iv) Combine the results above to construct the proof.

The external prover *pfvalid* helps us proving in steps i)-iii).

In Fig. 5, the function subHp converts a Hoare triple with the empty program to the corresponding logical sequent, and seqHp is the Hoare rule for appending programs which is similar to cnsHp. Both are provided in the library.

5.2 Loop Invariant and State Transition Model

We show properties 1 and 2 simultaneously, *i.e.*, make a proof term of Hoare triple "$Pre\,\{dsw\}\,Pos$", letting:

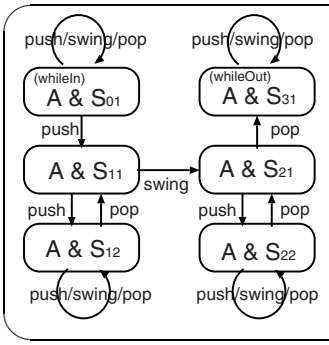
$$\begin{aligned} Pre &= @a\,(\neg m \wedge \Diamond_l b \wedge \Diamond_r c) \wedge @root\,\boldsymbol{\gamma}\,a \\ Pos &= @a\,(\ \ m \wedge \Diamond_l b \wedge \Diamond_r c), \end{aligned}$$

where the program variables a, b, and c are fresh and the modal formula $\gamma\, a$ is as in Fig. 3. Unlike property 3, finding a suitable loop invariant for proving the

```
Pre :: Pform
  = At_ a (not marked_ &&  not NULL_) && At_ root (eta a)
Pos :: Pform
  = At_ a (not marked_)

--   i) Show "Pre {init} Inv"
Inv :: Pform
  = At_ a (not marked_) && At_ p (not (eta a)) && At_ t (not (eta a))
initialize :: HTp init Pre Inv = external "pfvalid"

-- Cwhl is "!(p == Null) || !(t == Null || t.m == True)",
-- Cpsh is "!(t == Null || t.m == True)", and
-- Cswg is "p.s == False".
Gwhl :: Pform =                                      s2f Cwhl
Gpsh :: Pform =                          s2f Cpsh  && s2f Cwhl
Gswg :: Pform =      s2f Cswg  && not (s2f Cpsh) && s2f Cwhl
Gpop :: Pform = not (s2f Cswg) && not (s2f Cpsh) && s2f Cwhl

--  ii) Check the loop invariant
loopInv :: HTp loopBody (Gwhl && Inv) Inv
  = let invPush  :: HTp push  (Gpsh && Inv) Inv = external "pfvalid"
        invSwing :: HTp swing (Gswg && Inv) Inv = external "pfvalid"
        invPop   :: HTp pop   (Gpop && Inv) Inv = external "pfvalid"
    in cnsHp (iteHc (Gwhl && Inv)
                    invPush
                    (cnsHp (iteHc (not (s2f Cpsh) && Gwhl && Inv)
                                  invSwing
                                  invPop)
                           (nilHp idnt)))
             (nilHp idnt)

-- iii) Show "(not Cwhl && Inv) |- Pos"
exitwhl :: (not Gwhl && Inv) |- Pos
  = let exitwhl_ :: HTp [] (not Gwhl && Inv) Pos = external "pfvalid"
    in subHTp exitwhl_

--  iv) Main theorem
dswProp3 :: HTp dsw Pre Pos
  = seqHp initialize
          (cnsHp (whlHc Inv idnt
                        loopInv
                        exitwhl)
                 (nilHp idnt))
```

Lines beginning with "--" are comments.

Fig. 5. Verification of property 3 by Agda using *pfvalid*



$$
\begin{aligned}
A &= @a\,(\neg \mathtt{Null}) \wedge @p\,(\boldsymbol{\beta}\,\mathtt{Null}) \\
S_{01} &= @a\,(\neg m \wedge \Diamond_l b \wedge \Diamond_r c) \wedge (@p\,(\boldsymbol{\delta}\,a) \vee @t\,(\boldsymbol{\gamma}\,a)) \\
S_{11} &= @a\,(\ \ m \wedge \ \neg s \wedge \Diamond_r c) \wedge @p\,a \wedge @t\,b \\
S_{12} &= @a\,(\ \ m \wedge \ \neg s \wedge \Diamond_r c) \wedge @p\,(\boldsymbol{\beta}\,b) \wedge @b\,(\boldsymbol{\alpha}\,a) \\
S_{21} &= @a\,(\ \ m \wedge \Diamond_l b \wedge \ \ \ s) \wedge @p\,a \wedge @t\,c \\
S_{22} &= @a\,(\ \ m \wedge \Diamond_l b \wedge \ \ \ s) \wedge @p\,(\boldsymbol{\beta}\,c) \wedge @c\,(\boldsymbol{\alpha}\,a) \\
S_{31} &= @a\,(\ \ m \wedge \Diamond_l b \wedge \Diamond_r c) \wedge @p\,(\neg(\boldsymbol{\beta}\,a)) \\
Inv &= A \wedge (S_{01} \vee S_{11} \vee S_{12} \vee S_{21} \vee S_{22} \vee S_{31})
\end{aligned}
$$

Fig. 6. The state transition model

```
invPush :: HTp push (Gpsh && Inv) Inv
 = let (==>) (!pre, !pos :: List Pform) :: Set
         =  HTp push (Gpsh && A & dsj pre) (A & dsj pos)
       p01 :: [S01] ==> [S01, S11]  = external "pfvalid"
       p11 :: [S11] ==> [S12]       = external "pfvalid"
       p12 :: [S12] ==> [S12]       = external "pfvalid"
       p21 :: [S21] ==> [S22]       = external "pfvalid"
       p22 :: [S22] ==> [S22]       = external "pfvalid"
       p31 :: [S31] ==> [S31]       = external "pfvalid"
   in lem3 push Gpsh A unit unit
                        (p01 $ p11 $ p12 $ p21 $ p22 $ p31 $ nilz)
```

Fig. 7. Checking the transitions with "push"

Hoare triple is not a trivial task. To overcome the difficulty, we consider a state transition model as shown in Fig. 6, and define a p-formula *Inv* as the disjunction of the formulas which represent states in the model, where the schemata $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ are as in Fig. 3.

The p-formula *Inv* is clearly a loop invariant, if transitions in the model are all correct. Unlike the proof of property 3, we show the correctness of the state transition model, because the invariant formula is too large for our external prover *pfvalid* to check at a time. The proof in Fig. 7 witnesses the invariant with respect to the procedure push. It first proves, using *pfvalid*, the operation push changes each state in the model to some state in the model as shown in Fig. 6, and then concludes by combining these proofs using lem3 mentioned in Section 4.2. The "unit"s applied to lem3 are objects calling the internal provers to check the conditions, and the binary operator "$" constructs a sequence of Hoare proofs. Proofs for swing and pop are constructed similarly.

The construction of the state transition model is based on the following observation. We fix an arbitrary cell "a" reachable from the root-cell. Recall that the current cell is the cell that is held by variable p. At the beginning of the marking algorithm, a has not yet become the current cell. The state $A\&S_{01}$

Table 1. Running times of the external prover

	Properties 1,2						Property 3
initialize						10.55	0.50
	$A\&S_{01}$	$A\&S_{11}$	$A\&S_{12}$	$A\&S_{21}$	$A\&S_{22}$	$A\&S_{31}$	
invPush	28.19	0.88	5.36	1.00	6.97	5.02	0.84
invSwing	25.42	1.17	12.95	0.55	11.84	6.70	1.64
invPop	12.67	0.69	6.94	0.95	8.63	2.75	0.88
exits loop						2.98	0.63

represents the heap status at this time. At some time point, a becomes the current cell for the first time and a is marked. The state $A\&S_{11}$ represents the heap status at this time. Then, typically, p leaves a for a while. The heap status is represented by $A\&S_{12}$. The cell a becomes the current cell for the second time. The heap status is represented by $A\&S_{11}$ again. At this time, swing is operated and the swung field of a becomes true. The state $A\&S_{21}$ represents the heap status at this time. Again, typically, p leaves a for a while, with the heap status represented by $A\&S_{22}$, and goes back to a for the third and last time, with the heap status represented by $A\&S_{21}$. Then, pop is operated and the state $A\&S_{31}$ represents the heap status.

Although it is not straightforward to find exact formulas that are suitable for each state, we can use a trial and error approach. We select formulas that may represent the heap status based on the intuition and check whether they make the transitions correct. Due to the prover *pfvalid*, we can perform this procedure effectively.

The entire proof for the partial correctness of the DSW marking algorithm, which we have described, consists of 320 lines in the Agda language, including the encoded DSW algorithm, states in the model, and lemmas and their proofs. There are 25 calls to the external prover; Table 1 summarizes their running times measured in seconds. The tests were performed on a 1.33 GHz Core 2 Duo system running Windows XP.

5.3 Termination

We can apply the same approach as above to show that the DSW algorithm terminates, although we have not yet implemented a system to construct such proofs. We give a sketch of the proof. In this section, we describe heap statuses by modal formulas, not by p-formulas. We say that a pointer structure P satisfies a modal formula φ, if $K(P), s \models \varphi$ for any cell s in P. The notion of the strongest post-condition *spc* is defined based on this satisfaction relation.

We assume a special modality "o" in our modal language, called the *global modality*, which is always interpreted by $S \times S$ in $K(P)$. Furthermore, we have the *inverse modality* $\bar{f}$ for each modality $f \in \mathsf{F}$, that is interpreted by R_f^{-1}. $\mathrm{Sat}_\varphi P$ denotes the set $\{s \in S; K(P), s \models \varphi\}$ and we define the modal formulas $\mathrm{NI}_{cs}\,\varphi$ and $\mathrm{DE}_{cs}\,\varphi$ as:

$$\begin{aligned}\mathrm{NI}_{cs}\,\varphi &= \Box_o(\varphi \Rightarrow spc(\varphi, cs))\\ \mathrm{DE}_{cs}\,\varphi &= \Box_o(\varphi \Rightarrow spc(\varphi, cs)) \wedge \Diamond_o(\neg\varphi \wedge spc(\varphi, cs)),\end{aligned}$$

where P is a pointer structure on the cells S, φ a modal formula, and cs a basic block of a program. It is easy to prove the following lemma:

Lemma 1. *If formula* $\mathrm{NI}_{cs}\,\varphi$ *is valid,* $\mathrm{Sat}_\varphi\, P \supseteq \mathrm{Sat}_\varphi\,(cs\, P)$. *If formula* $\mathrm{DE}_{cs}\,\varphi$ *is valid,* $\mathrm{Sat}_\varphi\, P \supsetneq \mathrm{Sat}_\varphi\,(cs\, P)$.

Let $\varphi_1 = \neg m$, $\varphi_2 = \neg s$, and $\varphi_3 = \mu X.(p \vee \Diamond_{\bar{l}}(\neg s \wedge X) \vee \Diamond_{\bar{r}}(s \wedge X))$. The following formulas are valid:

$$\begin{array}{lll}\mathrm{DE}_{\mathrm{push}}\,\varphi_1, & \mathrm{NI}_{\mathrm{push}}\,\varphi_2, & \mathrm{NI}_{\mathrm{push}}\,\varphi_3,\\ & \mathrm{DE}_{\mathrm{swing}}\,\varphi_2, & \mathrm{NI}_{\mathrm{swing}}\,\varphi_3,\\ & & \mathrm{DE}_{\mathrm{pop}}\,\varphi_3.\end{array}$$

If the algorithm does not terminate, at least one of push, swing or pop is operated infinitely. Assume, for example, that swing occurs infinitely but pop is operated for a finite number of times. If we denote the i-th pointer structure at the top of the while loop by P_i, as per Lemma 1, $\mathrm{Sat}_{\varphi_2}\, P_i \supseteq \mathrm{Sat}_{\varphi_2}\, P_{i+1}$ for all $i \geq I$ for some natural number I and $\mathrm{Sat}_{\varphi_2}\, P_i \supsetneq \mathrm{Sat}_{\varphi_2}\, P_{i+1}$ for infinitely many i's, which is impossible. Similar arguments can be applied for other cases.

6 Conclusion and Future Work

In this paper, we showed that the correctness of the DSW marking algorithm can be proved using a framework based on a variant of the modal μ-calculus. For partial correctness, in particular, we built a system for constructing such proofs on top of the proof assistant Agda. Due to the external prover, we made the proofs very concise. We also showed that the termination of the DSW algorithm can be shown in our framework.

As future work, we plan to extend our system for termination proofs. First, the library for Agda should be extended to construct proofs, as in Section 5.3. Second, the command *pfvalid* needs to be extended to handle formulas with backward modalities and global modality. Backward modalities can be processed as shown in [11], and global modality can be handled in a straightforward manner. Although the procedure "*sat*" becomes incomplete in general due to the existence of backward modalities, it should be able to judge the validity of the six formulas in Section 5.3.

Although we have successfully verified the DSW algorithm, our framework is far from complete. The programming language lacks some important features such as function calls, especially recursive calls. The expressive power of the logic might be insufficient – the lack of existential quantifier is one of the problems. We are tackling these issues by trying to extend our framework.

Acknowledgments. This research was supported by the research project "Solving the description explosion problem in verification by means of structure transformation" in the CREST (Core Research for Evolution Science and Technology) program of the Japan Science and Technology Agency.

References

1. Agda Official Home Page, http://unit.aist.go.jp/cvs/Agda/
2. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
3. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center (1995)
4. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: Proc. Software Engineering and Formal Methods (SEFM 2005), pp. 190–199. IEEE Computer Society, Los Alamitos (2005)
5. Loginov, A., Reps, T.W., Sagiv, M.: Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 261–279. Springer, Heidelberg (2006)
6. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 121–135. Springer, Heidelberg (2003)
7. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
8. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM 10(8), 501–506 (1967)
9. Sekizawa, T., Tanabe, Y., Yuasa, Y., Takahashi, K.: MLAT: A tool for heap analysis based on predicate abstraction by modal logic. In: The IASTED International Conference on Software Engineering (SE 2008), pp. 310–317 (2008)
10. Tanabe, Y., Sekizawa, T., Yuasa, Y., Takahashi, K.: Pre- and post-conditions expressed in variants of the modal μ-calculus. CVS/AIST Research Report AIST-PS-2008-009, CVS/AIST (2008)
11. Tanabe, Y., Takahashi, K., Hagiya, M.: A decision procedure for alternation-free modal μ-calculi. In: Advances in Modal Logic (to appear, 2008)
12. Yang, H.: An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In: Proceedings of the 1st Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (2001)
13. Zappe, J.: Modal μ-calculus and alternating tree automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games. LNCS, vol. 2500, pp. 171–184. Springer, Heidelberg (2001)

Bounded Verification of Voting Software

Greg Dennis, Kuat Yessenov, and Daniel Jackson

Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
{gdennis,kuat,dnj}@mit.edu

Abstract. We present a case-study in which vote-tallying software is analyzed using a *bounded verification* technique, whereby all executions of a procedure are exhaustively examined within a finite space given by a bound on the size of the heap and the number of loop unrollings. The technique involves an encoding of the procedure in an intermediate relational programming language, a translation of that language to relational logic, and an analysis of the logic that exploits recent advances in finite model-finding. Our technique yields concrete counterexamples – traces of the procedure that violate the specification.

The vote-tallying software, used for public elections in the Netherlands, had previously been annotated with specifications in the Java Modeling Language and analyzed with ESC/Java2. Our analysis found counterexamples to the JML contracts, indicating bugs in the code and errors in the specifications that evaded prior analysis.

1 Introduction

First deployed for public elections in the Netherlands in 2004, the KOA Remote Voting System is an open-source, internet voting application written in Java. Intended to be used primarily by expatriots, KOA stands for the Dutch phrase "Kiezen op Afstand", which literally means "Remote Voting".

The KOA application contains a small vote-tallying subsystem that processes the ballot data and counts the votes. The vote-tallying module was developed independently of the rest of the application by the Security of Systems (SoS) research group at the University of Nijmegen, the developers of ESC/Java2 [1]. In building the module, SoS annotated their Java source code with specifications in the Java Modeling Language (JML) [2] and used their own ESC/Java2 tool to check the code against those specifications [3,4,5,6]. The code was also tested with the unit-testing tool *jmlunit* [7].

For our case-study, we used our own program analysis tool *Forge* [8] to check the vote-tallying code against the provided JML specifications. Our analysis found counterexamples to the JML contracts that evaded both the unit tests and ESC/Java2. Each counterexample is reported by Forge as a trace of the procedure under analysis, and each could indicate a bug in the code or an error in the specification. The purpose of the study was to explore the benefits and

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 130–145, 2008.

limitations of our approach, compare it to existing techniques like ESC and testing, and look for opportunities for synergy.

Our work is part of a larger, ongoing effort to develop a new technique for automatically checking object-oriented code against functional specifications. The basic idea, originating with Vaziri [9] and furthered by Taghdiri [10], involves translating a procedure in object-oriented code into a formula in the relational, first-order logic of Alloy [11] and invoking a finite model-finder to search for solutions to the formula. The model finder translates the relational formula to a boolean satisfiability (SAT) problem and hands it to a SAT solver. If the solver finds a solution, the model finder translates it to an instance of the relational formula, which is then converted into a trace of the procedure.

The approach, one of a larger class of analyses we refer to as *bounded verification*, examines all executions of a procedure up to some finite bound on the size of the heap and number of loop unrollings and reports counterexamples as actual procedure traces. It finds a counterexample if one exists within the provided bounds without issuing false alarms (given provisos described in Section 2.2); however, it will always miss bugs that require larger bounds for their detection. The effectiveness of the technique rests on the "small-scope hypothesis," the claim that many defects have small counterexamples, an idea consistent with our own experience and empirical evaluation [12,13].

Our prior work [13] introduced a more efficient translation of code to relational logic that exploited advances in the Kodkod model finder [14], the engine behind the latest version of the Alloy Analyzer. A proof-of-concept tool was demonstrated that worked for a small subset of Java and JML. This was a promising first step, but, as a tool applicable to non-trivial programs like KOA, it fell short. It was unable to handle common program features, including inheritance and arrays. Its more fundamental limitation, however, was a lack of support for fully modular analysis and datatype abstraction. Consequently, called procedures could not be summarized by specifications, and their code had to be inlined, causing the analysis to scale poorly.

To achieve full modularity, we turned our proof-of-concept into a more mature framework for bounded verification, a new component of which is an intermediate representation of code called the *Forge Intermediate Representation* (FIR). FIR is a simple relational programming language, capable of expressing imperative statements, declarative specifications, and relational abstractions, within a single small grammar. Our new approach is to translate both Java and JML into FIR and then apply our existing bounded verification technique to the resulting FIR program. The new framework is what made this case study practically feasible.

In the course of developing the bounded verification tool and technique, some questions were repeatedly raised, by ourselves and others. How useful is the technique? Is the "small-scope hypothesis" empirically justified? How does our technique compare to existing techniques? What are the key areas of future work on this project? The paper uses the results of the case study to help answer these questions.

2 Approach

Our early work introduced a new encoding of Java code in relational logic [13]. With the introduction of the new Forge Intermediate Representation, a full analysis of Java with Forge now involves a three-stage translation: from Java to FIR (Section 2.1); from FIR to relational logic (Section 2.2); and lastly, from relational logic to a boolean satisfiability problem (via the existing Kodkod tool).

2.1 From Java to FIR

The Forge Intermediate Representation (FIR), is an imperative, relational programming representation, capable of expressing both programs and their specifications. FIR is not a textual syntax; rather it consists of data structures assembled through an API. Like most intermediate representations, FIR was designed to be simple and uniform so as to be amenable to automatic analysis.

When we say FIR is "relational," we mean that every expression in the language evaluates to a *relation*, i.e. a set of *tuples*, where each tuple is a sequence of *atoms*. The arity of a relation (the length of its tuples) can be any positive integer. A set of atoms can be represented by a unary relation (a relation of arity 1), and a scalar by a singleton set.

The translation of object-oriented programs to FIR is based on a *relational view of the heap* [15], in which program data values are interpreted as relations. Specifically, types are viewed as sets; fields as functional relations; and local variables as singleton sets. To illustrate, we will refer to the Java example in Figure 1 and its resulting FIR translation in Figure 2. For simplicity, the example

```
class Birthday {
    /*@ non_null */ Month month;
    int day;

    //@ requires this.month.checkDay(d);
    //@ ensures this.day == d;
    void setDay(int d) {
        Month m = this.month;
        boolean dayOk = m.checkDay(d);
        if (dayOk) this.day = d;
    }
}

class Month {
    int maxDay;

    //@ ensures \result <==> (d > 0 && d <= maxDay);
    /*@ pure */ boolean checkDay(int d) { . . . }
}
```

Fig. 1. Birthday Example in Java with JML

```
00  domain Birthday, domain Month, domain Object
01  global month: Birthday → Month
02  global day: Birthday → Integer
03  global maxDay: Month → Integer
04  local this: Birthday, local d: Integer
05  local m: Month, local dayOk: Boolean
06
07  proc setDay (this, d) : ()
08    m = this.month;
09    dayOk = spec (dayOk ⇔ (d > 0 ∧ d ≤ m.maxDay));
10    if dayOk then day = day ⊕ (this → d) else exit;
```

Fig. 2. Translation of `Birthday.setDay` into FIR

translation ignores complexities arising from exception handling (which our tool does support).

For each concrete class in the code, the translation creates a corresponding FIR *domain* to represent the set of objects whose runtime type is that class. For the code in Figure 1, our translation declares three domains: Birthday, Month, and Object (Figure 2, Line 00). Two domains – Boolean and Integer – are built-in.

The translation maps each Java static type to a FIR type, either a domain or some combination obtained by the union or cross product of domains. The static types `Birthday` and `Month` are mapped to the FIR domains Birthday and Month respectively. Because every Java class is a subclass of `Object`, the translation maps the static type `Object` to the FIR union type Birthday ∪ Month ∪ Object.

The translation encodes each Java field as a global variable that maps members of the enclosing class to members of the field's type. For example, the Java field `month` is encoded as a FIR global month whose type is Birthday → Month (Line 01). Similarly, the translation creates global variables day: Birthday → Integer and maxDay: Month → Integer (Lines 02-03). The translation adds side constraints (not shown) that these binary relations are functions.

For each Java parameter and local variable in the method under analysis, the translation declares a local variable of the corresponding type, whose value is constrained to be a scalar. For the `setDay` method, the translation creates four FIR local variables: this of type Birthday, d of type Integer, m of type Month, and dayOk of type Boolean (Lines 04-05).

The FIR expression language is essentially the same as that offered by the Alloy modelling language [11]. Expressions can be built from any of the following: set operators (union, intersection, difference); relational operators (join, cross product, override, transitive closure); boolean, arithmetic, bitwise operators; set comprehension; and universal and existential quantification.

The result of translating the `setDay` Java method is the FIR setDay procedure, which has two inputs – this and d – and no outputs (Line 07). The procedure begins by assigning the FIR expression this.month to the local variable m (Line 08). Although the FIR statement looks nearly identical to its Java counterpart, the dot (.) operator in FIR stands for relational join, not field dereference. When

representing Java fields as functional relations and Java locals as singleton sets, field dereference can be encoded as relational join, because the join of a singleton set and a function will always yield a singleton.

The call to `checkDay` in the Java code has been encoded in FIR as a *specification statement* (Line 09) that embeds a declarative specification in imperative code [16,17]. FIR uses specification statements to facilitate modular analysis: once a procedure is found to meet a specification, calls to that procedure can be replaced with instantiations of its specification.

The specification statement in setDay is a FIR encoding of the JML contract for the `checkDay` method. Any variables on the left-hand side of the specification statement, in our example only dayOk, may be modified by the statement. On the right-hand side is a condition that the analysis establishes with demonic nondeterminism [18]; that is, the analysis will check the procedure for all executions that satisfy the condition.

An assignment to a field in Java is encoded using a relational override ($\oplus$) expression in FIR. The value of the FIR expression $\mathsf{day} \oplus (\mathsf{this} \rightarrow \mathsf{d})$ is the relation containing the tuple $(\mathsf{this} \rightarrow \mathsf{d})$ and any tuples in day that do not begin with this. Thus, the assignment $\mathsf{day} = \mathsf{day} \oplus (\mathsf{this} \rightarrow \mathsf{d})$ (Line 10) encodes the Java statement `this.day = d`. Note that the FIR assignment statement updates the *entire* value of the day relation (so that the symbolic execution mentioned below can compute a simple expression – not involving quantifiers or comprehensions – for the value of day at the end of the statement).

Datatype Abstractions. The translation from Java to FIR employs some useful datatype abstractions for common library collections, including sets, maps, and lists. Using abstractions of these collections in place of their concrete representations reduces the complexity of the resulting FIR program to be analyzed, thereby improving the performance of the analysis.

`Sets` are modelled abstractly by a binary relation whose domain is the Java `Set` objects and whose range is the elements in the set. A map is abstracted with a ternary relation that contains $(\mathsf{Map} \rightarrow \mathsf{key} \rightarrow \mathsf{value})$ tuples. Lists and arrays are abstracted by ternary relations containing $(\mathsf{List} \rightarrow \mathsf{index} \rightarrow \mathsf{value})$ tuples.

2.2 From FIR to Relational Logic

From a FIR procedure, the tool automatically obtains a formula $\mathsf{P(s, s')}$ in relational logic that constrains the relationship between a pre-state s and a post-state s' that holds whenever an execution exists from s that terminates in s'. A second formula $\mathsf{S(s, s')}$ is obtained from a user-provided specification, and its negation is conjoined to the first to obtain:

$$\mathsf{P(s, s')} \wedge \neg \mathsf{S(s, s')}$$

which is true exactly for those executions that are possible but that violate the specification.

The translation of procedural code to relational logic uses a symbolic execution technique that traverses each branch in the code, building a symbolic relational expression for each variable at each program point. Although our earlier work [13] presented the technique in the context of Java code, applying it to FIR is straightforward (in fact, much simpler due to FIR's relational structure).

In addition to the procedure and its specification, a client of Forge must provide a bound on the analysis consisting of:

- The number of times to unroll loops;
- The bitwidth limiting the range of FIR integers; and
- One *scope* for each domain, i.e. a limit on the number of its instances that may exist in any heap reached during execution.

Each of these limits results in under-approximation, eliminating possible behaviors but never adding behaviors. Thus, any counterexample generated will be valid – either demonstrating a defect in the code or a flaw in the specification. If a counterexample exists within the bound, one will be found, though defects that require a larger bound will be missed.

The chosen bound and the relational formula are handed to the Kodkod model finder. Kodkod translates the formula and the bound into a boolean satisfiability (SAT) problem, which it passes to an off-the-shelf SAT solver. If the solver finds a solution, Kodkod maps it to an instance of the relational logic formula, which Forge then maps to a counterexample trace of the original FIR procedure.

Soundness and Incompleteness. The only imprecision introduced by the translation from FIR to relational logic are the under-approximations given explicitly in the user-provided bound, so an analysis of FIR is sound in general and complete within the bounds. However, the translation from Java to FIR does not handle all of Java and also employs some optimizations, which introduces imprecision that may lead to spurious counterexamples and missed counterexamples.

One potential source of false alarms is the bounding of integers to a bitwidth less than that of Java integers. Consider an analysis in a bitwidth of 5 that produces a counterexample due to integer overflow. Because Java integers have a larger bitwidth, this counterexample does not represent an actual trace of the code. However, an overflow error in a small bitwidth, in our experience, usually indicate the presence of an analogous counterexample in a larger bitwidth. That is, if the code can overflow at a bit width of 5, it can probably overflow at 32.

Upon a method call, our analysis requires only the invariants of the receiver object hold. ESC/Java, in contrast, requires the invariants hold for every argument of a call. Exactly which invariants hold should upon method calls is an open area of research for modular analysis techniques generally.

The other sources of imprecision in the Java and JML translations are: lack of support for real number arithmetic, I/O, static initialization, reflection, `ArrayStoreExceptions`; incomplete support for String parsing; treatment of exceptions as singletons; unsound treatment of object equality; and unrolling of recursive specifications.

3 Analysis of KOA

At the time of the initial release of KOA in 2004, 47% of the core (non I/O) methods had been verified with ESC/Java2. The rest did not verify due to either non-termination of ESC's Simplify theorem prover, incompleteness in ESC, or unspecified "invariant issues." The code was also tested using *jmlunit*, which generated nearly 8,000 unit tests, all of which passed. Since that time, the code has been improved and further analyzed with ESC/Java2 [4].

Our analysis of the KOA vote-tallying software centered on eight classes, listed in Table 1, that form the core of its functionality. The `AuditLog` class logs the progress of the vote-tallying; `Candidate` records the tally of an individual candidate; `CandidateList` pairs a list of candidates in an election with a `CandidateListMetadata` that stores additional properties of the election; `District`, `KiesKring`, and `KiesLijst` are kinds of political district boundaries; and `VoteSet` records the cumulative tally for all candidates in the election. The *methods* column in Table 1 lists the number of methods analyzed in each class.

When Forge is applied to the methods of a class, the performance of the analysis depends not only upon the complexity of the code in the class but also upon the complexity of its specification, as well as the specifications of classes upon which that class depends. The *sloc* column in Table 1 gives the number of source lines of code in each class; *slocc* includes code and comment lines. Because JML is written inside Java comments, the slocc measures, albeit indirectly, the complexity of the class' code and specification. The *dslocc* is the *slocc* plus the number of lines of comment in classes upon which the class directly depends. The *dslocc/method* approximates the complexity of a modular analysis of a method within the class.

Table 1. Summary analysis statistics of each class. The means are calculated over the analyses of the methods within a class, not over successive analyses of the same class.

class	methods	sloc	slocc	dslocc	dslocc/ method	violations	mean scope	mean time (sec)
AuditLog	90	286	1237	1617	18.0	1	5.0	2.3
CandidateListMetadata	10	72	246	643	64.3	1	5.0	33.6
Candidate	12	103	363	1013	84.4	1	5.0	59.3
KiesKring	15	119	482	1272	84.8	5	5.0	249.7
District	6	53	163	543	90.5	0	5.0	18.5
KiesLijst	12	111	367	1432	119.3	4	5.0	104.6
CandidateList	13	130	493	1746	134.3	3	4.5	1416.8
VoteSet	11	113	400	2688	244.4	4	3.7	1783.9
Sum or *Mean*	**169**	**987**	**3751**	**10954**	*64.8*	**19**	*4.9*	*262.7*

As shown in the table, we applied Forge to a total of 169 methods of varying complexity across the eight classes. The *violations* column lists the number of methods that were found to violate their specification. A total of 19 specification violations were found. The experiments were run on a Mac Pro with two 3GHz Dual-Core Intel Xeon processors and 4.5GB RAM running Mac OS X 10.4.11. (Forge is single-threaded and so it did not take advantage of the multiple cores.) The code on which these analyses were conducted is, as of the time of this writing, the latest version available in its Subversion repository.

We initially analyzed each method in a scope of 5 instances of each type, an integer bitwidth of 4 (integers -8 to 7), and 3 loop unrollings. Most analyses completed quickly, but a few of the more complex methods exceeded our timeout of four hours. For those that timed-out, we progressively lowered the scope until the analysis completed within the time limit. The *mean scope* column in the table lists the average maximum scope in which the analysis successfully completed, and *mean time (sec)* is the mean time in seconds for a successful analysis. Note that these means are calculated over the analyses of the methods within a class, not over successive analyses of the same class. As shown in the table, the average analysis time is roughly correlated with the dslocc/method measure.

3.1 Specification Violations

Table 2 gives statistics on the 19 specification violations detected. The methods named `init` in the table are constructors. We inspected every violation and determined that none were false alarms (although theoretically possible, as noted at the end of Section 2.2). To evaluate the "small-scope hypothesis", for each violation found we progressively lowered the bound on the analysis (scope of each type, bitwidth of integers, number of loop unrollings) until the analysis no longer detected the counterexample. The minimum bound under which each counterexample is found is given in the last column of Table 2.

Every specification violation can be attributed to one of two causes: a bug in the code or an error in the specification. As outside observers, we can make educated guesses as to the cause, but classifying the violation with complete certainty requires knowing the programmer's intention. Does the specification accurately reflect the programmer's intention, in which case the violation indicates a bug in the code; or did the programmer err in transcribing his or her intention into the specification?

Specification errors themselves can be divided into two subcategories: *overspecification* and *underspecification*. A case of *overspecification* occurs when the specification of the method under analysis requires too much from the implementation – either its pre-condition is too weak or its post-condition is too strong. A case of *underspecification* occurs when the method under analysis calls a method whose specification provides too little to the caller – either the pre-condition of the called method is too strong or its post-condition is too weak. (Recall that our analysis, being fully modular, assumes that the specifications, not the implementations, of called methods define their behavior.)

Table 2. Specification violations: error classification and minimum bound (scope/ bitwidth/unrollings) necessary for the error's detection

class	method	error	min bound
CandidateListMetadata	init	under	1 / 3 / 1
KiesKring	addDistrict	bug	1 / 3 / 1
VoteSet	addVote(String)	over	1 / 3 / 1
KiesLijst	clear	over	1 / 3 / 3
AuditLog	getCurrentTimeStamp	over	2 / 1 / 1
Candidate	init	under	2 / 3 / 1
CandidateList	addDistrict	under	2 / 3 / 1
CandidateList	addKiesLijst	over	2 / 3 / 1
CandidateList	init	over	2 / 3 / 1
KiesKring	addKiesLijst	bug	2 / 3 / 1
KiesKring	init	under	2 / 3 / 1
KiesKring	make	under	2 / 3 / 1
KiesLijst	addCandidate	over	2 / 3 / 1
KiesLijst	compareTo	bug	2 / 3 / 1
KiesLijst	make	over	2 / 3 / 1
VoteSet	addVote(int)	over	2 / 3 / 1
VoteSet	validateKiesKringNumber	over	2 / 3 / 1
VoteSet	validateRedundantInfo	over	2 / 3 / 1
KiesKring	clear	over	2 / 3 / 3

Specification errors, while not "bugs" per se, are still important to address. For example, there may be methods whose correctness depends on the overspecification of a called method. Fixing the overspecification may, therefore, reveal latent bugs in dependent methods. In contrast, a case of underspecification doesn't pose an immediate problem, but it could allow a bug to be introduced in the future. That is, the underspecified method's implementation could be changed at a later date in a way that still satisfies its contract, but causes dependent methods to fail.

As shown in Table 2, of the 19 violations found by Forge, we believe 3 to be due to buggy code, 11 due to overspecification, and 5 due to underspecification. From our follow-up "smallest scope" analyses of each violating method, we found that every violation would have also been found in a scope of 2, a bitwidth of 3, and 3 loop unrollings.

In fact, all but two of the violations required only 1 loop unrolling, the exceptions being `KiesKring.clear` and `KiesLijst.clear`. Both `clear` methods contain loops with if-statements in the body of the loop, and 3 unrollings were necessary to cover all paths. Additionally, four violations needed only the minimal scope of 1 and one violation was found in the minimal bitwidth of 1.

A minimal bitwidth of 3 (integers from -4 to 3) was needed for nearly every analysis, because some static array fields in the code were required to be of at least length 2. Lowering the bitwidth to 2 would allow a maximum integer of only 1. These arrays were not required to be fully populated, however – they could contain null elements – so their minimal length requirements did not in turn affect the minimal scope necessary to detect violations.

3.2 Example Violations

In this section, we present and discuss a sample of four specification violations detected by our analysis, two of which we've classified as bugs, one overspecification, and one underspecification. For brevity, some of the code excerpts and JML specifications shown below have been simplified.

(a) KiesLijst.compareTo [bug]. The code for the method is a correct implementation of the `compareTo` method in `Kies`*`Kring`*, not `Kies`*`Lijst`*. This is likely a copy-and-paste error:

```
class KiesLijst {
  public int compareTo(final Object an_object) {
    if (!(an_object instanceof KiesKring)) {
      throw new ClassCastException();
    }
    final KiesKring k = (KiesKring) an_object;
    return number() - k.number();
  }
}
```

The `instanceof` check and the initialization of local variable `k` should refer to `KiesLijst`, not `KiesKring`.

(b) KiesKring.addDistrict [bug]. The `KiesKring` class stores an array of districts in the `my_districts` field and a count of the number of districts in the `my_district_count` field. The specification for `KiesKring` includes an invariant that the count is equal to the number of non-null entries in the array:

```
private final /*@ non_null @*/ District[] my_districts
private byte my_district_count;
//@ invariant my_district_count == (\sum int i; 0 <= i && i < my_districts.length;
//@                                             (my_districts[i] != null) ? 1 : 0);

boolean addDistrict(final /*@ non_null @*/ District a_district) {
  if (hasDistrict(a_district)) {
    return false;
  }
  my_districts[a_district.number()] = a_district;
  my_district_count++;
  return true;
}
```

Each district has a number that is used as its index in the array. The `hasDistrict` method returns true when the `my_districts` array contains a district with the same number and name as its argument. Thus, if the `a_district` argument has the same number but a *different* name than a district already in the array, the method will overwrite an existing district and increment the district count in violation of the invariant. The district count should only be updated only if there is no existing district at that index.

This violation might be classified as a specification error if the programmer forgot an invariant prohibiting two districts from having the same number but different names. The rest of the code does not appear to rely on such an invariant, however. Indeed, the `District.equals` method checks for equality by comparing not only the number but also the name.

(c) VoteSet.addVote [overspecification]. This method suffers from overspecification in the form of a missing precondition. Note that it invokes the method `Candidate.incrementVoteCount`:

```
class VoteSet {

  final void addVote(final int a_candidate_code) throws IllegalArgumentException {
    if (!(my_vote_has_been_initialized && !my_vote_has_been_finalized)) {
      throw new IllegalArgumentException();
    }
    final Candidate candidate = my_candidate_list.getCandidate(a_candidate_code);
    candidate.incrementVoteCount();
    candidate.kiesLijst().incrementVoteCount();
  }
}

class Candidate {

  //@  requires my_vote_count < AuditLog.getDecryptNrOfVotes();
  //@  modifies my_vote_count;
  //@  ensures my_vote_count == \old(my_vote_count + 1); final
  int incrementVoteCount() { . . . }
}
```

As shown, `incrementVoteCount` has a precondition that the number of votes for the candidate be less than a preset number, but `addVote` does not ensure this condition. We believe the programmer erred in not including the inequality constraint in the precondition of `addVote`. It is also possible that the programmer intended `addVote` to be robust when the inequality is false, in which case we would re-classify this violation as a bug.

(d) KiesKring.init [underspecification]. The post-condition of the `KiesKring` constructor invokes an underspecified `KiesKring.name` method:

```
//@ requires a_kieskring_name.length() <= KIESKRING_NAME_MAX_LENGTH;
//@ ensures number() == a_kieskring_number;
//@ ensures name().equals(a_kieskring_name);
private /*@ pure @*/ KiesKring(final byte a_kieskring_number,
                               final /*@ non_null @*/ String a_kieskring_name) {
  my_number = a_kieskring_number;
  my_name = a_kieskring_name;
}

//@ ensures \result.length() <= KIESKRING_NAME_MAX_LENGTH;
/*@ pure non_null @*/ String name() { return my_name; }
```

The specification of the constructor claims that calling `name()` in the post-state yields a string equal to the `a_kieskring_name` argument, and the constructor does indeed assign the argument to the `my_name` field. However, even though the implementation of `name` returns the `my_name` field, its specification says merely that it returns some string whose length is less than a fixed constant. Thus, the post-condition of `name` does not induce a strong enough axiom to establish the post-condition of `init`. Indeed, an implementation of `name` that always returns the empty string would satisfy its weak specification, but would clearly cause `init` to violate its own specification.

4 Discussion

Did bounded verification prove to be a useful technique? Prior to our case study, the KOA software had been the subject of rigorous development. The code, written according to a "verification-centric methodology" [5], had been checked with ESC/Java2 and unit-testing. Despite these prior efforts, our technique found that 19 of the 169 methods analyzed violate their specification.

The effort in conducting the study did not prove particularly burdensome. The KOA developers had already written the JML specifications, which was presumably a time-consuming task, but if full functional correctness is required, it seems unlikely that writing a full specification can be avoided. The remaining effort was choosing a bound on each analysis and inspecting the counterexample traces detected, both of which we found to be straightforward.

For most methods, we were pleased with the runtime and scalability of the analysis. For others, we were disappointed that their analysis in a scope of 5 did not complete within a reasonable amount of time. The poor performance of these methods demonstrates clear room for improvement in our technique and helps direct our future work on this project, as discussed below.

Does the case study lend support to the "small scope hypothesis"? That we found several errors in a small scope lends some support to the hypothesis that in practice many errors have small counterexamples. But it tells us nothing about proportions; it is possible that the software analyzed is riddled with bugs beyond the bounds we checked. Ideally, one would increase the scope step by step until all errors are revealed, and then determine their distribution. This is infeasible, but we were able to do this in miniature by increasing the scope until the analysis became intractable, and by noting (Table 3.1) the smallest scope in which a given error is revealed. That increasing the scope from 2 to 5 reveals no additional errors suggests that at least within that range the small scope hypothesis holds.

For many violations, detection required fields (array fields in particular) to be pre-populated. To detect the violation of `KiesKring.addDistrict` (Section 3.2b), for example, the array of districts needed to store a district that was distinct from the district given as an argument. Thus, finding the error requires a scope of at least 2 (in this case 2 districts).

We were somewhat surprised that the minimal bitwidth stayed as high as 3. After all, some methods didn't even use integers. As explained at the end of Section 3.1, the minimal bitwidth of 3 was due to length requirements on some array fields in the code. Even though a lower bound may not violate the precondition of the method under analysis, it still may prevent the existence of a valid pre-state heap configuration.

How does our technique compare to unit testing? On one level, it may seem surprising that our bounded verification technique revealed specification violations missed by unit testing. After all, bounded verification is conceptually a form of testing, in which all tests up to some small size are executed. However, two properties of our technique distinguish it from unit testing in key ways. One

important difference is that our technique is modular and, therefore, can detect problems due to underspecification of called methods, while unit testing cannot.

The major difference, however, is one of coverage. The voting code was subject to nearly 8,000 unit tests, which on its face sounds like a large number of tests to generate and run. Our technique, however, by leveraging SAT-solving technology, is capable of analyzing thousands, if not millions, of scenarios of every method individually.

Despite these differences, it still came as a surprise that unit testing did not detect the buggy `KiesLijst.compareTo` method discussed in Section 3.2a. Perhaps the static type of the `compareTo` parameter being `Object` (not `KiesLijst`) caused the testing tool to only feed the method arguments of *runtime* type `Object`. In these cases, the buggy version of the method would behave correctly by raising a `ClassCastException`.

To catch the bug in `hasDistrict`, discussed in Section 3.2b, a unit-test would need to first populate the pre-state with a non-empty array of districts, and then pair the pre-state with a district argument with the same number but a different name from an existing district in the array. Revealing such a bug requires a higher level of coverage than can be expected from traditional unit-testing.

How does our technique compare to ESC/Java2? It is difficult for us to determine why ESC/Java2 failed to detect the specification violations found in our study, and, unfortunately, the authors of the ESC study were unable to provide additional information in this regard. We know that 53% of the KOA methods did not verify successfully with ESC/Java2, so perhaps all 19 violations fell into this category. Or perhaps some of the 19 did verify but, due to unsoundness in ESC, were actually faulty. Kiniry, Morgan, and Denby [19] detail the sources of unsoundness and incompleteness in ESC/Java2.

Two examples where the unsoundness of ESC may have played a role are the analysis of KOA methods `KiesKring.clear` and `KiesLijst.clear`. ESC/Java2 examines only one unrolling of each loop, but our case study found that at least three unrollings were necessary to detect the overspecification of those methods.

What areas for future work appear worthwhile? There are a number of changes that we plan to make to Forge to improve its performance. One area for improvement highlighted by this study is the need to exploit generics in the Java source code. The presence of a Java `Map` in the code introduces a ternary relation whose second and third columns range over the universe of all objects, an expensive relation to encode in SAT. Generics were not used in KOA, but when provided, an improved translation to FIR could exploit the type arguments for the keys and values of a `Map`, significantly reducing the state space needed to represent the ternary relation and dramatically improving the scalability of the analysis.

Beyond technical enhancements, we see further opportunities for hybrid techniques that combine bounded verification and theorem proving in a way that leverages the advantages of each. For example, a tool could perform a bounded verification analysis in a progressively higher bound until some timeout is reached and then apply an automated prover like ESC. Or perhaps an automated prover

could use bounded verification to dispose of only the less tractable portions of the verification conditions, such as those involving alternating quantifiers.

Lastly, we will continue to support developers building translations from other high-level languages to FIR, to make them amenable to our analysis. An automatic translation from C is already in development by the System Engineering Lab at Toshiba, and we would greatly welcome similar efforts.

5 Related Work

In addition to the three tools discussed in this paper – ESC/Java2, jmlunit, and Forge – there are a number of other tools for checking Java code against JML specifications. Some, such as the KeY System [20], exploit theorem proving technology. Another promising tool is Kiasan [21], a new symbolic execution and test-case generation framework for Java and JML. It would be of great value to see these tools applied to the KOA code, learn what errors the find or fail to find, and understand the effort required to apply them.

This work extends our prior work [13], which itself builds on the work of Vaziri [9], whose Jalloy tool analyzed Java code via a translation to the Alloy modeling language. Dolby and Vaziri have since offered some optimizations to their technique [22] that might offer benefits to our own analysis.

The Forge Intermediate Representation is similar to other available programming languages and representations, first and foremost being the Alloy modeling language [11] and the subset of its logic that is accepted by the Kodkod model finder [14]. In a sense, one could view FIR as an "imperative Alloy."

DynAlloy [23] is an extension to Alloy which allows one to specify and check dynamic properties of relational models. Unlike FIR, Dynalloy is still a declarative representation, not an imperative programming language and offers a different approach to encoding Java in relational logic that Frias, et al, are pursuing.

FIR is not the first programming notation based on sets and relations. An early example is SETL [24], a high-level programming language founded on set theory and set operations. Another example is the Crocopat relational manipulation language [25]. Unlike these, FIR supports declarative constraints via specification statements in the code. Also, FIR supports object instantiation, whereas Crocopat presumes a finite, predefined universe of objects.

Less similar in semantics but more similar in purpose to FIR is BoogiePL [26], the intermediate programming language accepted by Boogie, a static verifier for Spec#. Like FIR, it offers a full specification language including quantifiers, but does not use relations as its fundamental datatype, so it would not be a convenient representation to encode in relational logic.

Acknowledgements

This research was funded in part by the National Science Foundation under grant 0541183 (Deep and Scalable Analysis of Software) and the Toshiba Corporate

Research and Development Center. We would like to thank the Digital Security group at the Radboud University Nijmegen and KindSoftware at the University of Dublin for making KOA available.

References

1. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: PLDI 2002, pp. 234–245 (June 2002)
2. Leavens, G., et al.: Java Modeling Language, http://www.jmlspecs.org
3. Jacobs, B.: Counting votes with formal methods. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116. Springer, Heidelberg (2004)
4. Fairmichael, F.: Full verification of the KOA tally system University College Dublin. Bachelor's Thesis (March 2005)
5. Kiniry, J., Morkan, A., Cochran, D., Fairmichael, F., Chalin, P., Oostdijk, M., Hubbers, E.: The KOA remote voting system: A summary of work to date. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661. Springer, Heidelberg (2007)
6. Kiniry, J.: Formally counting electronic votes (but still only trusting paper). In: ICECCS 2007, pp. 261–269 (July 2007)
7. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)
8. Dennis, G., Yessenov, K.: Forge website, http://sdg.csail.mit.edu/forge/
9. Vaziri, M.: Finding Bugs in Software with a Constraint Solver. PhD thesis. MIT (February 2004)
10. Taghdiri, M.: Inferring Specifications to Detect Errors in Code. In: 19th Int. Conf. on Automated Software Engineering (ASE 2004), Linz, Austria(September 2004)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
12. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating Small Scope Hypothesis (2003) (MIT CSAIL unpublished manuscript)
13. Dennis, G., Chang, F.S.H., Jackson, D.: Modular verification of code with SAT. In: ISSTA 2006, Portland, Maine (July 2006)
14. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424. Springer, Heidelberg (2007)
15. Jackson, D.: Object models as heap invariants. Essays on Programming Methodology, 247–268 (2000)
16. Morgan, C.: The specification statement. In: ACM Trans. Program. Lang. Syst, vol. 10, pp. 403–419. ACM Press, New York (1988)
17. Morgan, C.: Programming from Specifications. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
18. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
19. Kiniry, J.R., Morkan, A.E., Denby, B.: Soundness and completeness warnings in ESC/Java2. In: SAVCBS 2006, New York, pp. 19–24 (2006)
20. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007)
21. Deng, X., Lee, J., Robby,: Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: ASE 2006, Washington, DC, pp. 157–166. IEEE Computer Society, Los Alamitos (2006)

22. Dolby, J., Vaziri, M., Tip, F.: Finding bugs efficiently with a SAT solver. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593. Springer, Heidelberg (2007)
23. Frias, M., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 442–451. Springer, Heidelberg (2006)
24. Dewer, R.: The SETL Programming Language (1979)
25. Beyer, D.: Relational programming with Crocopat. In: Proceedings of the 28th International Conference on Software Engineering (May 2006)
26. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft (2005)

Expression Decomposition in a Rely/Guarantee Context

Joey W. Coleman

School of Computing Science,
Newcastle University,
NE1 7RU, UK
j.w.coleman@ncl.ac.uk

Abstract. This paper describes a technique of expression decomposition which allows the use of rely/guarantee development rules that do not assume atomic expression evaluation. This decomposition provides a means of addressing the fact that the logical meaning of expressions relative to a single state and the semantic evaluation of expressions in a fine-grained concurrent language do not provide the same results; in particular, the former results in a single value whereas the latter can result in many possible values. Rely/guarantee development rules tend to depend on the logical meaning of expressions in cases where they are used; expression decomposition identifies where it is safe to do so, and provides some tools for where it is not.

1 Introduction

In [Jon07] Jones poses a challenge to create a formal method for concurrent system development –Atomicity Refinement– which is based on the "fiction of atomicity". This "fiction" is the idea that software can be designed as though it will be executed in an interference-free context, then through careful refinement and data reification, the portion of that software which requires atomicity may be split into pieces which can be safely executed in an environment where interference is present.

Unfortunately, no such general method exists to date and we will not provide one in this paper. What we will provide, however, is a means of safely decomposing expressions and identifying properties of the components in such a way that they may be used with a greater degree of finesse in a rely/guarantee reasoning framework.

The technique of expression decomposition is presented here in the context of rely/guarantee reasoning because it characterizes the behaviour of the expected interference. This technique is adaptable to any method so long as it provides some means of characterizing the behaviour of interference; as such, this technique can be viewed as a contribution towards Jones' Atomicity Refinement.

The semantic and logical framework which provides the context for this work is described in Section 2, and it includes structural operational semantics, rely/guarantee reasoning, and an extension to the initial structural operational

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 146–160, 2008.

semantics which incorporates elements of rely/guarantee reasoning. Section 3 introduces our notion of expression decomposition and explains its use in the rely/guarantee framework. We then conclude the paper in Section 4.

2 Semantic and Logical Framework

This paper assumes the use of rely/guarantee reasoning on top of a well-defined language semantics. We set aside concerns regarding the definedness of expressions in this paper because the extra conditions required to ensure definedness add little to the core ideas in this work. In this section we will outline a particular semantic model of a language and a few rely/guarantee development rules.

2.1 Semantic Model

The results developed in Section 3 are predicated on the language having two main properties: first, that it has some form of shared-variable concurrency so that interference is an issue; and second, that the concurrency is fine-grained, meaning in particular that the values of variables in an expression may be interfered with during the evaluation of said expression. These properties result in a language which is similar, in spirit, to commonly used languages which support concurrent execution models. We will model this language using structural operational semantics (SOS) [Plo81, Plo04] and VDM-style notation [Jon90]. The language model is a subset of the one used in [CJ07] and [Col08]; here we have selected only the portions of the larger model relevant to the task at hand.

Expressions in the language consist of values (including Boolean values) of type *Value*, a set of identifiers, *Id*, monadic operations on expressions, *Monad*, and dyadic operators on pairs of expressions, *Dyad*. Together, these four types are combined to give the general expression type, *Expr*.

$$Expr = Value \mid Id \mid Monad \mid Dyad \mid \ldots$$

The abstract syntax of the *Monad* types are defined by VDM records:

$$\begin{aligned} Monad &:: op : MonadOp \\ &\quad\ a : Expr \end{aligned}$$

The *MonadOp* type includes operators appropriate for the construct. The *Dyad* type (and similar possible expression constructs for more operands) is a straightforward extension of the *Monad* type, and will be left implicit.

We are assuming that the overall expression type has an associated well-formedness predicate, though we will not define it here. This predicate can be defined in the usual way, and an example can be found in [Col08]. As part of the purpose of well-formedness predicates is to ensure the type correctness of expressions, we will also assume that all expressions are well-typed. We also assume that all expressions used in this paper are well-defined.

Since this language models shared-variable concurrency, we use the type Σ to model the memory store.

$$\Sigma = Id \xrightarrow{m} Value$$

Instances of the memory store are referred to as states throughout this paper, and are denoted by σ.

Expression evaluation is modelled using the transition relation $\xrightarrow{e}$.

$$\xrightarrow{e} \in \mathcal{P}\left((Expr \times \Sigma) \times Expr\right)$$

We use the parentheses in the type definition of the transition relation as an informal indication of how we actually use write a given transition.

The order of evaluation of the components of an expression is taken to be arbitrary; no particular order may be assumed. Should a specific order of evaluation be required it can be emulated through a series of separate statements, assigning values to temporary values as needed.

This relation is defined by a small set of inference rules. Reading the value of a variable from the state is given by the *Id-E* rule.

$$\boxed{\textit{Id-E}}\ \frac{id \in Id}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

This rule reads the value of a variable in a single step and, as such, we take the reads and writes of a variable to be atomic, regardless of the size of the value being written.

Monadic operations use two rules to model their behaviour: *Monad-Eval*, which allows one evaluation step on the operand; and *Monad-E*, which performs the specified operation on the completely-evaluated operand.

$$\boxed{\textit{Monad-Eval}}\ \frac{(a, \sigma) \xrightarrow{e} a'}{(\textit{mk-Monad}(op, a), \sigma) \xrightarrow{e} \textit{mk-Monad}(op, a')}$$

$$\boxed{\textit{Monad-E}}\ \frac{a \in \textit{Value}}{(\textit{mk-Monad}(op, a), \sigma) \xrightarrow{e} [\![op]\!](a)}$$

The use of Strachey brackets in *Monad-E* simply indicates that we wish to take the logical meaning of the specified operator and apply it to the value given.

Evaluation rules for dyadic operations would be structured in a similar manner to those for the monadic operator. To maintain the arbitrary ordering of the dyadic expression we use the same non-deterministic choice between rules as is found in *Par-L* and *Par-R*, below. Examples of semantic rules for the *Dyad* construct can be found in [CJ07] and [Col08].

The rules which define the $\xrightarrow{e}$ transition relation are structured such that any single step will do precisely one of two things: either replace an identifier with its value from the supplied state, or replace a *Monad* with the result of applying its operation. This minimal behaviour, combined with the concurrency present in the language model, allows for very fine-grained interference during evaluation.

We will only define the syntax and rules for the statements which are directly relevant to Section 3. Specifically, we will define a construct which provides concurrency so as to show the sort of concurrency model we are interested in,

and we will define a construct for conditional execution, which gives us a practical connection to expression evaluation; we also note the presence of an assignment construct as the source of interference in the language, but we will not provide a formal semantics for assignment in this paper. The overall type of statements is *Stmt*, which is (partially) defined to be

$$Stmt = Assign \mid Par \mid If \mid \ldots \mid \mathbf{nil}$$

Statement execution is modelled using the $\xrightarrow{s}$ transition relation, and this relation describes the basic behaviour of the language system which we are interested in.

$$\xrightarrow{s} \in \mathcal{P}\left((Stmt \times \Sigma) \times (Stmt \times \Sigma)\right)$$

The assignment construct, *Assign*, has the usual definition and semantics.

$$\begin{array}{rcl} Assign :: id & : & Id \\ e & : & Expr \end{array}$$

Assignment evaluates an expression over multiple steps and when the expression is reduced to a value it writes the value into the state in a single step.

The parallel construct, *Par*, provides interleaving concurrency in the execution of the two statements it contains.

$$\begin{array}{rcl} Par :: left & : & Stmt \\ right & : & Stmt \end{array}$$

Three rules are used to model the behaviour of the parallel construct. The first, *Par-E*, is a simple bookkeeping rule, used to eliminate the construct when both of its components have finished executing. The second rule, *Par-L*, allows the *left* component of the parallel construct to make a single step; the third rule, *Par-R*, is similar to the second rule, allowing a single step of the *right* component.

$$\boxed{Par\text{-}E}\ \frac{}{(mk\text{-}Par(\mathbf{nil}, \mathbf{nil}), \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma)}$$

$$\boxed{Par\text{-}L}\ \frac{(left, \sigma) \xrightarrow{s} (left', \sigma')}{(mk\text{-}Par(left, right), \sigma) \xrightarrow{s} (mk\text{-}Par(left', right), \sigma')}$$

$$\boxed{Par\text{-}R}\ \frac{(right, \sigma) \xrightarrow{s} (right', \sigma')}{(mk\text{-}Par(left, right), \sigma) \xrightarrow{s} (mk\text{-}Par(left, right'), \sigma')}$$

In this definition of the behaviour of the parallel construct the choice between taking a step on the *left* component or the *right* component is non-deterministic. This model of concurrency ensures that all possible interleavings of steps of the branches are modelled by this language definition. It is the interaction between the type of concurrency provided by this parallel construct and the *Expr* semantics described above that generate the characteristic features of fine-grained semantics: small observable steps and the sort of interleaving between separate threads as shown here.

The next construct –representing conditional execution– is included to motivate and give the framework for the results in Section 3.

$$\begin{array}{rcl} If :: e & : & Expr \\ body & : & Stmt \end{array}$$

The behaviour of the *If* construct is not unusual and is essentially the same as similar constructs found in commonly used languages.

Evaluation of the test expression, e, is done through the *If-Eval* rule.

$$\textit{If-Eval}\ \frac{(e,\sigma) \xrightarrow{e} e'}{(\textit{mk-If}(e,\textit{body}),\sigma) \xrightarrow{s} (\textit{mk-If}(e',\textit{body}),\sigma)}$$

This rule performs one evaluation step per statement step, so its interaction with the rules for the parallel construct allow for the state object to change during the overall evaluation of the expression.

Once the expression is fully evaluated to a Boolean value, one of two rules gives the transition which disposes of the *If* construct. The first, *If-F-E*, simply eliminates the construct when the test expression has evaluated to **false**.

$$\textit{If-F-E}\ \frac{}{(\textit{mk-If}(\mathbf{false},\textit{body}),\sigma) \xrightarrow{s} (\mathbf{nil},\sigma)}$$

The second rule, *If-T-E*, handles the case where the evaluation of the test expression evaluated to **true**. In this case the rule drops the *If* construct, leaving behind the *body* component in its place.

$$\textit{If-T-E}\ \frac{}{(\textit{mk-If}(\mathbf{true},\textit{body}),\sigma) \xrightarrow{s} (\textit{body},\sigma)}$$

Note that the wording regarding evaluation in the preceding paragraphs is precise: the rules are concerned with whether there is a **true** or **false** value in the *If* construct, not whether the logical meaning of an expression relative to the current state is **true** or **false**.

2.2 Rely/Guarantee Reasoning

The particular rely/guarantee framework used in this paper is similar to the ones found in Jones' thesis [Jon81] and a case study by Jones and Collette [CJ00]. The primary difference is that we do not assume that expression evaluation is atomic.

There is an important distinction to be made regarding atomic evaluation of expressions and expressions that are unaffected by interference. The former, atomic evaluation, is applicable in situations where the underlying language has a semantic model in which expressions are evaluated completely in one step. This presumption still requires that a developer carefully finesse situations where interference is unavoidable, but it does eliminate the possibility of interference *during* expression evaluation. Expressions which are unaffected by interference, however, have the much stronger property which allows those expressions to be treated as though they were in a non-concurrent context. Determining whether or not an expression is unaffected by interference requires knowledge of the context in which the expression will be evaluated; a simple example of an expression not affected by interference is when the interference does not alter any of the variables in the expression. Rely/guarantee reasoning is aimed at situations where the

expressions may be affected by interference and assuming a sequential context is questionable at best.

Satisfaction of a rely/guarantee specification is indicated by logical sentences of the form

$$(P, R) \vdash \mathit{prog}\ \mathbf{sat}\ (G, Q)$$

where

- P is a pre condition, a predicate over states, which characterizes the initial states for which the program will run correctly;
- R is a rely condition, a relation over states, which characterizes the maximum interference which the program can tolerate while still executing correctly;
- *prog* is the program which satisfies the constraints G and Q under the assumptions P and R;
- G is the guarantee condition, also a relation over states, which characterizes the maximum interference which the program may produce;
- and Q is the post condition, another relation over states, which constrains the final states which the program may produce relative to the initial states.

It is important to note that the pre and rely conditions are assumptions about the overall environment in which the program will be executed; they are not constraints which must be satisfied by the program. Thus, the sentence above can be read as: "Assuming an initial state which satisfies P and interference bounded by R, we can show that the behaviour of *prog* is bounded by G and the execution of *prog* will terminate and produce a state which, when paired with the initial state, satisfies Q". Termination is an important element of rely/guarantee reasoning and is required property for any program which claims to satisfy a rely/guarantee specification. Related to termination, we assume that all processes which can run will eventually be allowed to do so.

The rely, guarantee, and post conditions are all of the same type.

$$R, G, Q \in \mathcal{P}\,(\Sigma \times \Sigma)$$

One program's guarantee condition may become a part of another program's rely condition (and vice versa), so we require the rely and guarantee to be the same type. Rely and guarantee conditions are assumed to be both reflexive and transitive in this paper. The post condition may be both intransitive and irreflexive as it is not a constraint at the same level of granularity.

To illustrate the sort of reasoning and abstraction used by rely/guarantee reasoning, consider the rule used for the parallel construct.

$$\boxed{\textit{Par-I}}\ \frac{\begin{array}{c}(P, R \vee G_r) \vdash \mathit{left}\ \mathbf{sat}\ (G_l, Q_l) \\ (P, R \vee G_l) \vdash \mathit{right}\ \mathbf{sat}\ (G_r, Q_r) \\ G_l \vee G_r\ \Rightarrow\ G \\ \overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^*\ \Rightarrow\ Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Par}(\mathit{left}, \mathit{right})\ \mathbf{sat}\ (G, Q)}$$

Though this rule does not illustrate the problem we face with expression evaluation, it does illustrate the ability of the reasoning framework to abstract the

behaviour of a program component, and then use that abstraction to reason about the satisfaction of other components. The cross-dependency of the *left* and *right* branches of the parallel on each others' guarantee conditions actually eases the task of reasoning about the branches on their own. As we know that a guarantee condition is an outer bound on the behaviour of a program, and that any satisfying program can always tolerate less interference than is present in its assumed rely condition, we can then proceed to find programs that satisfy the specifications for the branches without concern for the details of how the opposite branch works.

In a rely/guarantee framework that assumes atomic expression evaluation the development rule for the *If* construct is straightforward.

$$\boxed{\textit{If-Atomic-Eval-I}}\ \frac{(P \wedge e, R) \vdash \textit{body}\ \mathbf{sat}\ (G, Q) \qquad \overleftarrow{P} \wedge \neg\, \overleftarrow{e}\ \Rightarrow\ Q}{(P, R) \vdash \textit{mk-If}(e, \textit{body})\ \mathbf{sat}\ (G, Q)}$$

The *If-Atomic-Eval-I* is not a valid development rule for a language with fine-grained concurrency, such as the one given in Section 2.1. The problem is that an expression may *evaluate* to **false** even though its logical meaning is **true** (or vice versa) relative to every state used during the evaluation process. We can consider the expression $x = x$ as an example: if the value of x changes between reads of the variable then evaluation will produce **false**.

We can alter the *If-Atomic-Eval-I* rule so that it is valid for a language with fine-grained concurrency with a single naïve addition to the rule.

$$\boxed{\textit{If-Ident-I}}\ \frac{R\ \Rightarrow\ I_{Vars(e)} \qquad (P \wedge e, R) \vdash \textit{body}\ \mathbf{sat}\ (G, Q) \qquad \overleftarrow{P} \wedge \neg\, \overleftarrow{e}\ \Rightarrow\ Q}{(P, R) \vdash \textit{mk-If}(e, \textit{body})\ \mathbf{sat}\ (G, Q)}$$

The newly-added first antecedent requires that interference does not alter the values of the variables in the expression. The unfortunate effect of adding this antecedent, however, is that the evaluation of the test expression must now happen in a context which is essentially sequential and, to exacerbate the difficulty of using this rule in a development, the antecedent also requires that the interference be so constrained for the whole execution of the body. While this may be appropriate to some situations, this is a much tighter constraint than the previous assumption of atomic evaluation.

An alternate naïve formulation of *If-Atomic-Eval-I* which is valid for the language of Section 2.1 involves removing instances of the expression, e, from the antecedents of the rule.

$$\boxed{\textit{If-Opt-I}}\ \frac{(P, R) \vdash \textit{body}\ \mathbf{sat}\ (G, Q) \qquad \overleftarrow{P}\ \Rightarrow\ Q}{(P, R) \vdash \textit{mk-If}(e, \textit{body})\ \mathbf{sat}\ (G, Q)}$$

This rule is clearly at the opposite extreme than *If-Ident-I*: instead of tightly constraining the interference so that the expression may be used in the rule, it

simply does not use the expression in the rule at all. However, this rule is only usable in circumstances where the execution of the body is completely optional with respect to the satisfaction of the post condition. It is unlikely that this rule could be useful for the development of a program, except possibly for debugging statements.

These two rules for the conditional construct –*If-Ident-I* and *If-Opt-I*– illustrate the practical problem faced by the usual logical (atomic) treatment of expressions when the actual underlying language semantics dictate a non-atomic mechanism. In the rely/guarantee framework, simply treating an expression as a predicate[1] is of little use unless we also know that the variables contained in the expression will not be affected by interference during evaluation.

2.3 Extended Semantics

The last two pieces of semantic machinery that we introduce are useful for reasoning about the behaviour of a program during the design process. They mix the transition relations of Section 2.1 with a rely condition as defined in Section 2.2.

Given a rely condition, R, representing the possible interference from the environment, we can define a pair of new transition relations based on the transition relations given earlier. One relation, $\xrightarrow{s,R}$, models the execution of a program in that environment, and the second, $\xrightarrow{e,R}$, models the evaluation of an expression in that environment.

The first transition is $\xrightarrow{s,R}$, and is defined by two rules: *S-Step* and *S-R*.

$$\boxed{\textit{S-Step}}\ \frac{(s,\sigma) \xrightarrow{s} (s',\sigma')}{(s,\sigma) \xrightarrow{s,R} (s',\sigma')}$$

$$\boxed{\textit{S-R}}\ \frac{[\![R]\!](\sigma,\sigma')}{(s,\sigma) \xrightarrow{s,R} (s,\sigma')}$$

The *S-Step* rule is simply an inclusion of the entire $\xrightarrow{s}$ transition relation. The *S-R* rule allows the state object to change in any way consistent with the rely condition; this rule effectively simulates the (worst-case) possible interference which another program running in parallel could generate, if that program's guarantee condition was the same as the given rely condition.

We have a similar pair of rules for expression evaluation under interference which define the $\xrightarrow{e,R}$ transition relation. These rules can be thought of as focusing the $\xrightarrow{s,R}$ transition relation in on just a particular expression evaluation.

$$\boxed{\textit{E-Eval}}\ \frac{(e,\sigma) \xrightarrow{e} e'}{(e,\sigma) \xrightarrow{e,R} (e',\sigma)}$$

[1] Either directly in the case of Boolean expressions, or indirectly via constructions such as $e = \langle \textit{value} \rangle$ or $e \in \langle \textit{value-set} \rangle$.

$$\boxed{E\text{-}R}\ \frac{[\![R]\!](\sigma, \sigma')}{(e, \sigma) \xrightarrow{e,R} (e, \sigma')}$$

The *E-Eval* rule allows for a single evaluation step, and the *E-R* rule allows for interference from the environment to change the state object.

These semantic relations are useful both for reasoning about expression evaluation during program development and for creating soundness proofs of rely/guarantee rules. They are not, however, easy to use directly within a development rule because they deal with a whole expression rather than easily characterizable pieces of the expression.

Our extended semantics incorporates interference (as constrained by a rely condition) directly into the model and allows us to reason about the results of execution (and evaluation) as affected by interference. In this paper we are only concerned with the results of evaluation, and the extended semantics gives a simple way of accessing them. An alternative approach –described by de Roever et al. as "reactive sequences" [dR01]– incorporates interference through a trace-based semantics where "gaps"[2] are used to indicate interference. It is not difficult to see how a reactive sequence could be generated using the extended semantics given here.

3 Expression Decomposition

This section describes a method which can mitigate the problem encountered in Section 2.2. It is possible to use the structure of expressions to create rely/guarantee development rules which are more useful for use in software development at the cost of a small loss in generality. The essence of the idea is to identify the unstable (i.e. subject to interference) portions of an expression and split them from the stable (i.e. unaffected by interference) portions; having done that, we can create development rules that then make use of the separate portions of the expression in a way that reflects the nature of those portions.

3.1 Expression Transformation

Consider the expression, e, such that

$$e \triangleq x < min(y, z)$$

Semantically, e will evaluate to **true** when the value read for x is less than the values read for both of y and z; logically, e is equivalent to **true** for any state where the value of x is less than the values of both y and z. Once again, the difference in wording reflects the tangible difference between the semantic evaluation of an expression and the logical meaning of an expression.

Let us also assume a rely condition, R, such that

$$R \triangleq x = \overleftarrow{x} \wedge y = \overleftarrow{y} \wedge z \leq \overleftarrow{z}$$

[2] Places in the sequence where the i^{th} final state and $i + 1^{st}$ initial state differ.

that is, we will assume that x and y are unaffected by interference, but that z may be monotonically decreased by the environment. From R we can deduce that $R \Rightarrow I_{\{x,y\}}$ — the rely condition is an identity relation with respect to the variables x and y. This means that both x and y are stable variables in our terminology. Since we know that the rely condition allows changes to z, this variable is unstable.

This expression, e, is an unstable expression: the evaluation of this expression depends upon the contained unstable variable. This form of the expression is not useful in the context of a rely/guarantee development rule. We can rewrite e as e', where

$$e' \mathrel{\underline{\triangle}} x < y \wedge x < z$$

While e and e' are logically equivalent, we do not know that they will have the same behaviour when evaluated under interference. For e' to be usable as a replacement for e, we also need to know that they both have the same set of possible results when evaluated under interference; that is, we need to know that the following holds.

$$\forall \sigma \in \Sigma \cdot \{v \mid v \in \mathit{Value} \wedge (e, \sigma) \xrightarrow{e,R} \!\!\!\!\rightarrow\!\!* (v, \sigma')\} = \{v \mid v \in \mathit{Value} \wedge (e', \sigma) \xrightarrow{e,R} \!\!\!\!\rightarrow\!\!* (v, \sigma')\}$$

The aim of this property is to preserve the complete set of possible results under interference, not to ensure that the resulting expression is still valid in the context of the development, nor to allow a refinement step that reduces the non-determinism in evaluation. Ensuring the contextual validity of the decomposed expression requires that it be used in a rely/guarantee development rule; indeed, the need to decompose an expression may arise because the initial expression was not usable with the development rules at hand. Refinement steps can, as usual, be applied after a decomposed expression has been found to be valid in development process.

We note two observations which are valid for any sound decomposition of e into e'. First, the two expressions must be logically equivalent, i.e.

$$\forall \sigma \in \Sigma \cdot [\![e]\!](\sigma) = [\![e']\!](\sigma)$$

and, second, the addition or removal of constant expressions and stable variables (so long as logical equivalence is preserved) does not affect the set of possible results.

Also, we propose as a guideline that for each unstable variable in the original expression, there should be the same number of instances of that variable in the transformed expression. This guideline is pragmatic in intent: its purpose is to preserve the inherent non-determinacy in the expressions, as manifested by the number of reads of unstable variables.

This guideline is neither necessary nor sufficient to ensure a correct decomposition. On one hand, adding instances of an unstable variable in the conjunction of the expression $z = z \vee \mathbf{true}$ to e as defined earlier would result in an expression which preserves the set of possible resulting values: the addition is a constant expression regardless of interference. Critically, however, if we consider

another pair of expressions, $w - w = 0$ and $w \times w \geq 0$, where w is an unstable variable constrained to non-negative numbers, we can see that these two expressions do not have the same set of possible results under interference even though they do have the same number of instances of w. The guideline is not without its merits, however: following it prevents relatively simple failures such as the transformation of $2z$ into $z + z$ — a pair for which it is obvious that they do not in general produce the same set of possible results.

We call this process "expression decomposition" as it results in an expression for which we can not only identify the independent terms in the structure of the expression, but we can also use these terms independently in a rely/guarantee rule. In the case of e', we have two terms: one of which, $x < z$, is unstable and not directly usable; however, the other term, $x < y$, is stable and thus we can use it directly at the logical level.

3.2 Extending the Logical Framework

We have now seen that we can decompose an expression into terms which can be classified as stable and unstable; it will be useful to indicate these properties in the rely/guarantee development rules, so we now define two predicates for this purpose.

The first predicate indicates that an expression is composed entirely of stable variables relative to a given rely condition.

$$\mathit{StableExpr}\colon \mathit{Expr} \times \mathit{Rely} \to \mathbb{B}$$

As noted earlier, a stable expression is just one for which all of its constituent variables are unaffected by interference; for the language used here, this simply means that the rely condition acts as an identity relation relative to the variables in the expression. A corollary of this is that the logical meaning and semantic evaluation of any stable expression are equivalent.

Expressions which yield a constant result regardless of the state they are evaluated in, and regardless of interference, could be included in the definition of *StableExpr*. However, this sort of trivial case is not a source of difficulty in reasoning about the evaluation of expressions under interference. For example, the expression $(x = x) \vee \mathbf{true}$, even if x is subject to interference, will always evaluate to **true**.

The second predicate indicates that the expression contains only a single unstable variable, and only a single instance of that particular variable (again, relative to a given rely condition).

$$\mathit{SingleUnstableVar}\colon \mathit{Expr} \times \mathit{Rely} \to \mathbb{B}$$

The purpose of the *SingleUnstableVar* predicate is to identify a particular class of unstable expressions which can be used in the development rules.

The evaluation of expressions for which this predicate holds will be *as though* the evaluation took place in a single state. Unfortunately, we do not know *which*

state the evaluation is equivalent to, but we gain a lot from what we do know. So, if $SingleUnstableVar(e, R)$ holds, then we have the property

$$\forall \sigma \in \Sigma \cdot \{v \mid v \in Value \wedge (e, \sigma) \xrightarrow{e,R} * (v, \sigma')\} = \{[\![e]\!](\sigma') \mid \sigma' \in \Sigma \wedge \sigma R \sigma'\}$$

That is, we know that the set of values which can be produced by evaluation under interference are exactly the same as the set of values which are the logical meanings of the expression with respect to the set of states reachable through interference.

Use of the *SingleUnstableVar* predicate forms a bridge between the semantic evaluation of an unstable expression and the logical meaning of the expression. Though it only does so in a limited way, we will show that it is sufficient to build development rules that directly address a class of unstable expressions.

3.3 Application to the Development Rules

Now that we have a decomposition mechanism for expressions and two predicates which allow us to use properties in the decomposed expressions, let us see how they may be used in the rely/guarantee development rules. Here we present a new development rule for the *If* construct; in Section 3.4 we set our earlier example expression, e, into a larger example.

Our new development rule for the *If* construct takes advantage of the additional information provided using decomposed expressions and the predicates defined above.

$$\textit{If-I}\ \frac{\begin{array}{c}(P \wedge s, R) \vdash body\ \mathbf{sat}\ (G, Q) \\ StableExpr(s, R) \\ SingleUnstableVar(u, R) \\ \neg \overleftarrow{u} \wedge R \ \Rightarrow\ \neg u \\ \overleftarrow{P} \wedge \neg (s \wedge u) \ \Rightarrow\ Q\end{array}}{(P, R) \vdash mk\text{-}If(s \wedge u, body)\ \mathbf{sat}\ (G, Q)}$$

This development rule for the *If* construct is much more involved than the previous versions; most of this additional complexity is a direct result of reasoning about non-atomic expression evaluation.

Starting with the consequent of the rule, we first note that the test expression has structure: two conjoined terms, s and u. These terms correspond to the stable and unstable portions of the test expression, and this is recorded in the second and third antecedents. Since the s term of the test expression is stable, we are free to use it at the logical level without any constraints, and thus it appears in the pre condition of the first antecedent. This allows the body of the *If* to depend upon at least a part of the test expression.

The fourth and fifth antecedents are concerned with the situation where the test expression evaluates to **false** (in contrast to the first antecedent, which is concerned with the **true** case). These antecedents depend on evaluation of the unstable portion of the test expression, u, happening as though it were done in

a single state; this requirement is met by the third antecedent. So, the fourth antecedent ensures that if u is logically equivalent to **false**, then interference cannot result in u being logically equivalent to **true**; since the third antecedent means that evaluation happens as though a single state were used consistently, this constraint applies to evaluation as well. The fifth antecedent addresses the **false** evaluation case directly, requiring that the post condition be satisfied for any pair of states where the pre condition holds for the initial state, and the whole test expression is logically equivalent to **false** in the final state. By not requiring the whole test expression to be **false** in the initial state we allow for situations where the test expression was equivalent to **true** in the initial state, but interference caused u to become **false** before the semantic evaluation read the unstable variable.

There is an odd quirk of this rule in practise in that it permits the body of the construct to be run in the case where the unstable variable in u is read in a state where u is equivalent to **true**, and interference then causes u to become equivalent to **false** at some point afterwards but before the body completes (or even starts) execution. This behaviour still meets the specification because the first antecedent only assumes that the stable portion of the expression, s, was true.

Finally, the *If-I* rule we note that it can be simplified into either of *If-Ident-I* or *If-Opt-I* by setting either of u or s to **true**, respectively. Taking the first case, if we set u to be **true**, we find that the third and fourth antecedents of *If-I* become irrelevant and the remainder of the antecedents simplify to precisely the antecedents of *If-Ident-I*. The other case works out in a similar manner.

3.4 Example Use of the Development Rule

With the new *If-I* rule in mind, let us now look at an example of its use. We will consider a fragment of the FINDP example from Owicki's thesis [Owi75]. The overall example is concerned with the task of finding the lowest index in an array which satisfies some predicate, *pred*; the example presented here is concerned only with the portion of the task which decides whether or not to actually test the current index.[3] This task is developed in a concurrent manner to show interference between sub-tasks of the search.

Let x denote the current index under consideration for the current thread, y denote the upper bound of the search for the current thread, and z denote the upper bound of the search for all alternate threads. When the minimum of y and z is greater than the maximum index of the array then no index satisfying *pred* has yet been found; however, when the minimum of y and z is a valid index in the array then this value is the lowest known index to satisfy *pred*. The x and y variables are considered to "belong" to the current thread, while z does not; this allows us to assume the rely condition defined earlier, R, as the rely condition for this example.

The local thread should only test for satisfaction of the current index against *pred* if the current index is less than the minimum of y and z; this is precisely

[3] It is presumed that *pred* is expensive to evaluate, and so is to be avoided if possible.

the expression defined earlier as e. We can see that neither the two naïvely defined rules for the *If* construct can be applied in this situation: *If-Ident-I* is not applicable because the rely condition is unsuitable; and *If-Opt-I* is precluded as we need to be certain that the body will be executed in some situations.

We cannot use the *If-I* rule with e, however, so we must decompose it into e'. The decomposed expression matches both the structure of and the constraints on the test expression in the *If-I* rule because $x < y$ is stable relative to the rely condition and $x < z$ has only a single unstable variable. The fourth antecedent –that once the unstable expression becomes logically equivalent to **false** interference never generates a state in which it is logically equivalent to **true**– is satisfied by $x < z$ and the rely condition because R only allows for z to monotonically decrease. The fifth antecedent is also satisfied: if the test expression is logically equivalent to **false** then x was not less than either of y or z, and because of this it was not necessary to check to see if x satisfied the predicate.

4 Conclusions

The core of this paper is concerned with the use of the fine structure of expressions in situations where interference is unavoidable. The use of an expression's structure allows us to use properties of portions of that expression in the rely/guarantee development rules, even in situations where the only thing that can be inferred about the overall expression is that its behaviour under evaluation is different than the logical meaning of the expression in any given state. We have also provided a set of constraints to decompose expressions so that their structure can be accessed directly. This concern is one of the major issues which will need to be addressed by a method of atomicity refinement and the material presented here gives a possible approach to it.

As we only give a single rule which applies these ideas to the task of developing conditional constructs, it would be useful to provide rules for other constructs. There is a development rule in this style for a looping construct, *While*, provided in [Col08], but as yet there are no general rules for constructs such as assignment.

The predicates introduced in Section 3.2, *StableExpr* and *SingleUnstableVar*, are aimed specifically at expressions for which there is only a single unstable variable. This is a limiting constraint on the work presented here, and though it is not hard to see that this work could be extended to expressions which conjoin multiple terms which satisfy *SingleUnstableVar*, it is still an open question as to how to characterize terms which have multiple unstable variables.

Acknowledgements. The technical content of this paper has benefited from discussions with Cliff Jones, both during the writing of this paper and during the thesis which precedes it.

The author would like to thank the referees for their comments as they have improved this paper immensely. In particular, one referee provided a counterexample to the guideline in Section 3.1; a previous version of this paper made

stronger claims regarding that guideline, though this version does not. The author is currently working on a notion of refinement –suggested in a related comment by another referee– which looks to handle the counterexample and provide a more constructive mechanism for expression decomposition.

The author also gratefully acknowledges research funding for the EPSRC project "Splitting (Software) Atoms Safely" and the EPSRC platform grant on "Trustworthy Ambient Systems".

References

[CJ00] Collette, P., Jones, C.B.: Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In: Foundations of Computing, pp. 277–307. MIT Press, Cambridge (2000)

[CJ07] Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. Journal of Logic and Computation 17(4), 807–841 (2007)

[Col08] Coleman, J.W.: Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics. PhD thesis, Newcastle University, Newcastle Upon Tyne (January 2008)

[dR01] de Roever, W.-P.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. In: Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, New York (2001)

[Jon81] Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis. Oxford University (June 1981); Printed as: Programming Research Group, Technical Monograph 25

[Jon90] Jones., C.B. (ed.): Systematic Software Development Using VDM, 2nd edn. Prentice- Hall, Inc., Englewood Cliffs (1990)

[Jon07] Jones, C.B.: Splitting atoms safely. Theoretical Computer Science 375(1-3), 109–119 (2007); (Festschrift for John C. Reynolds's 70th birthday)

[Owi75] Owicki, S.S.: Axiomatic Proof Techniques for Parallel Programs. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, USA, 75-251 (1975)

[Plo81] Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University (1981)

[Plo04] Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming, 60–61, 17–139 (2004)

A Verification Approach for System-Level Concurrent Programs*

Matthias Daum, Jan Dörrenbächer, Mareike Schmidt, and Burkhart Wolff

Saarland University, Computer Science Dept.
66123 Saarbrücken, Germany
{md11,jandb,mareike,wolff}@wjpserver.cs.uni-sb.de

Abstract. Though the verification of operating systems is an active research field, a verification method is still missing that provides both, the proximity to practically used programming languages such as C *and* a realistic model of concurrency, i. e., a model that copes with the granularity of atomic operations actually used in a target machine.

Our approach serves as the foundation for the verification of concurrent programs in C0 – a C fragment enriched by kernel communication primitives – in a Hoare-Logic. C0 is compiled by a verified compiler into assembly code representing a cooperative concurrent transition system. For the latter, it is shown that it can actually be executed in a true concurrent way reflecting the C0 semantics.

1 Introduction

Industrial-strength software analysis and verification has advanced in recent years through the introduction of static analysis techniques, model checking as well as automated and interactive theorem proving. However, many techniques are working under restrictive assumptions that limit their applicability to complex (embedded) system software such as operating system kernels, low-level device drivers or microcontroller code.

In this paper, we present a theorem-proving based method that can cope with unbounded data structures (buffers, stacks), in contrast to model-checking techniques limited to state spaces that have usually to be finite and small.

Based on Leinenbach and Petrova's [1] formally verified compiler from a C variant – called sequential C0 – into assembly language, we provide the foundation of an abstract verification technique for concurrent programs on top of a particular microkernel. The main problem is to establish a relation between a true concurrent execution on a low-level machine and a cooperative concurrent model, which is suitable for verification. With *cooperative concurrent execution*, we refer to a conventional sequential execution except when basic communication primitives are called; after their call, a process may resume with a non-deterministically updated state (although the possible places where memory

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

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 161–176, 2008.

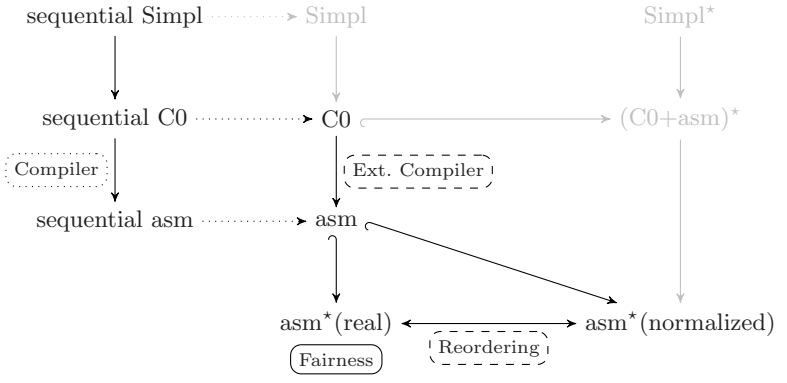


Fig. 1. Overview of the Language Stack

might change is actually very restricted in our model). By *true concurrent*, we mean an execution where a process may be arbitrarily interrupted and resumed.

In our paper, we label sequential models with the index $^{\text{seq}}$, cooperative concurrent models with $^{\text{cc}}$ and true concurrent ones with $^{\text{tc}}$. Fig. 1 presents the "grand picture" of our language stack. Subsequently, we briefly introduce the languages together with the underlying machines:

The core of the stack is the language "C0", which is described by the small-step execution model $A^{\text{cc}}_{\text{C0}} = (\mathcal{S}_{\text{C0}}, \mathcal{S}^0_{\text{C0}}, \delta^{\text{cc}}_{\text{C0}})$. The states $\mathcal{S}_{\text{C0}}$ of the automaton are the *configurations* of the small-step semantics, i. e., pairs $(prog, mem)$ of a C0-program *prog* yet to be executed, and the current memory *mem*. As usual, the initial states $\mathcal{S}^0_{\text{C0}}$ are a subset of $\mathcal{S}_{\text{C0}}$, and the transition relation $\delta^{\text{cc}}_{\text{C0}}$ is a binary relation over states. The language is a straight-forward extension of Leinenbach's "sequential C0" by kernel communication primitives. In this model, it is assumed that concurrency can *only* occur at these primitives, and that the result of actions of the process environment is only visible in form of non-deterministic changes in (a restricted) area of *mem*.

The assembly language "asm" is described by the machine $A^{\text{cc}}_{\text{asm}}$ and executes the same operations as Leinenbach's "sequential asm" machine; however, $A^{\text{cc}}_{\text{asm}}$ is extended by kernel primitives related to operations of our microkernel VAMOS. This machine is more concrete than the abstract $A^{\text{cc}}_{\text{C0}}$ machine because it has no strict separation between code and data, and can therefore handle self-modifying code.

We associate "asm*(real)" with an operational VAMOS machine A^{tc}_{v}. This machine is a true concurrent model with a number of assembly processes and an explicit scheduler. There is an interface to external devices like a timer, a hard-disk or a console. Its transition relation is defined to simulate the $A^{\text{cc}}_{\text{asm}}$ machine. Another operational kernel machine, $A^{\text{tc}}_{\text{vc}}$, is associated with "asm*(normalized)", which has no explicit scheduler but behaves otherwise like A^{tc}_{v}.

Schirmer [2] has proven a Hoare-Logic for his imperative language "Simpl" sound and complete wrt. its semantics. He provides a whole verification environment comprising a verification condition generator as a means for effective

verification of Simpl programs. Furthermore, he has shown an embedding of sequential C0 into "sequential Simpl". Albeit his work could be easily adapted to our (cooperative concurrent) C0, we do not discuss this further and rather focus on the foundation for its use in a concurrent setting.

As main contributions of this paper, we provide:

- The extension of the sequential C0 semantics and the sequential assembly semantics with suitable kernel communication primitives;
- An extended compiler correctness theorem: the translation of C0 (including the kernel primitives) to assembly preserves the operational semantics,
- A formal proof in Isabelle/HOL that the operational kernel machine $A_{\mathrm{v}}^{\mathrm{tc}}$ provides fair scheduling between assembly processes,
- A reordering theorem allowing for the reinterpretation of execution traces of the $A_{\mathrm{v}}^{\mathrm{tc}}$ machine in terms of $A_{\mathrm{asm}}^{\mathrm{cc}}$.

Thus, we provide a method that is on the one hand concrete enough for the verification of user-mode device drivers and similar operating-system components. On the other hand, we can indeed reason in a sequential, process-local style via a Hoare-Logic over concurrent programs. However, this reasoning can only establish partial correctness of process systems; in the future work section, we will therefore discuss the implications of our work for machines such as "(C0+asm)*", i. e., concurrent process systems, and its abstraction to Hoare-Logic "Simpl*" enabling the proof of system-global liveness properties.

2 Background

2.1 Fundamentals of Our Microkernel

In this section, we sketch the features of Vamos and explain the fundamental access-control mechanism that establishes the process roles "privileged process" and "device driver".

Our microkernel Vamos performs the following tasks: (a) enforcement of a minimal access control, (b) process management, (c) memory management, (d) priority-based round-robin scheduling, (e) support for user-mode device drivers, and (f) inter-process communication (IPC). Processes can control these tasks via the kernel's application binary interface (ABI). Table 1 lists the kernel calls that constitute the ABI.

Most kernel calls are reserved for so-called *privileged processes*. Thus, only a privileged process can bring up new processes or kill existing ones, alter the memory consumption of processes, change their scheduling parameters, or control the registration of device drivers. However, any process might use the IPC mechanism. Thus, the privileging serves as a minimal access-control mechanism. We presume that the privileged processes constitute the user-mode parts of the operating system and implement a more sophisticated access control. Non-privileged processes may then use IPC in order to request from the privileged processes that they call the kernel on behalf of the non-privileged ones.

Table 1. Application binary interface of the VAMOS kernel

Kernel Call	Description
Access Control	
`set_privileged` [p]	add a process to the set of privileged processes
Process Management	
`process_create` [p]	create a new process from a memory image
`process_clone` [p]	copy an already existing process
`process_kill` [p]	kill a process
Memory Management	
`memory_add` [p]	increase the amount of virtual memory for a process
`memory_free` [p]	decrease the amount of virtual memory for a process
Scheduling Mechanism	
`chg_sched_params` [p]	change scheduling parameters
Device Driver Support	
`change_driver` [p]	(de)register a process as a driver for a set of devices
`enable_interrupts` [d]	re-enable a set of interrupts after their successful handling
`dev_read` [d] / `dev_write` [d]	communicate with a certain device
Inter-Process Communication	
`ipc_send`	send a message to another process
`ipc_receive`	receive a message from another process
`ipc_request`	send a message and immediately wait for a reply
`change_rights`	manipulate IPC rights
`read_kernel_info`	receive information from the kernel

[p] call is reserved for privileged processes [d] call is reserved for device drivers

When VAMOS boots, it launches one single process, the *init process*. This process is privileged and has to set up the required servers of the operating system, start and register the device drivers, and possibly run initial applications.

A *device driver* is a user process, which is designated for the communication with certain devices. Only if a process is registered as a driver for a particular device, it may place read or write requests from or to that device, respectively. Moreover, the device driver is notified of interrupts from that device.

2.2 On a Correct Compiler

In this section, we summarize work of Leinenbach and Petrova [1,3]. We often deal with structured values, which we define by enumerating the components in prose, e. g., "a value x consisting of two components *this* and *that*". We refer to a single component with a dot, e. g., $x.this$ refers to component *this* of value x. An update of this component is denoted by $x\,[this := q]$.

The Language Sequential C0. ANSI C has a complex and highly underspecified semantics. However, low-level kernel programs such as drivers explicitly *use* properties of a particular compiler on a target hardware, for example, its little-endian-ness or a particular atomicity of assembly operations. They can therefore not be verified based only on the vague ANSI C semantics. In our approach, we

constrained ourselves to the C-like imperative language *C0*, which has sufficient features to implement low-level software, but which is interpreted by a more concrete semantics. C0's most important limitations compared to ANSI C are:

- Expressions must be free of side effects and do not contain function calls,
- There are no implicit type conversions, especially not from arrays to pointers,
- Pointers are strongly typed and must not point to functions or stack variables (i. e., there are neither void pointers nor pointer arithmetic), and
- Low-level data types (like unions and bit fields) and control-flow statements (like switch and goto) are not supported.

Syntax. C0 supports fundamental types, aggregate types and pointers. The first category comprises Booleans, 8-bit-wide characters, as well as signed and unsigned 32-bit integers. Aggregate types in C0 are arrays and structures. Pointers may point to all types of data but not to functions.

Expressions are variable names and literals. Moreover, if e and i are expressions and n is a component name, array access $e[i]$, access to structure components $e.n$, dereferencing $*e$, and the "address-of" operation $\&e$ are expressions. Additionally, C0 supports the usual unary and binary operators.

Finally, C0 supports statements for assignments, dynamic memory allocation, sequential composition, conditional and repeated execution, inline assembly, function calls and returns from functions.

Small-step Semantics. For lack of space, we can only glance at the semantics. C0 programs are statically represented by the program environment Γ, which comprises a symbol table of global variables, a type-name environment, and a function table. The symbol table is a list of pairs of variable names and types. The type-name environment maps type names to types. The function table maps function names to functions, which are represented by a tuple consisting of (a) a symbol table for the function's parameters, (b) a symbol table for the stack variables, (c) the function's return type, and (d) a statement representing the function body.

The dynamically changing state s_{C0} of a C0 program in execution comprises:

- The remaining program $s_{\mathrm{C0}}.\mathit{prog}$, and
- The current state of the program variables $s_{\mathrm{C0}}.\mathit{mem}$.

In the following sections, we assume an evaluation function *get-val* for the look-up, and an update function *set-val* for the manipulation of a certain variable in the memory. We refer to the value of expression e in state s_{C0} by $\mathit{get\text{-}val}(s_{\mathrm{C0}}, e)$. If we update the left-value l in state s_{C0} with some expression u, we denote the resulting configuration by $\mathit{set\text{-}val}(s_{\mathrm{C0}}, l, u)$.

The transition relation $\delta_{\mathrm{C0}}^{\mathrm{seq}}$ of this semantics is deterministic, i. e., a partial function.

The Target Assembly Language. The assembly semantics was developed for the RISC processor VAMP [4]. It abstracts from the paging mechanism of the

processor and employs a linear memory model. An assembly state s_{asm} consists of the following components:

- The normal and the delayed program counters, $s_{\text{asm}}.pc$ and $s_{\text{asm}}.dpc$, respectively, implementing the delayed branch mechanism.
- The general-purpose register file $s_{\text{asm}}.gpr \in \{0,\ldots,31\} \to \{0,1\}^{32}$.
- The memory size $s_{\text{asm}}.V$ measured in pages of 4096 bytes. It defines the set of available memory addresses: $VA(s_{\text{asm}}) = \{a \mid a < s_{\text{asm}}.V \cdot 4096\}$
- The byte-addressable linear memory $s_{\text{asm}}.vm \in VA(s_{\text{asm}}) \to \{0,1\}^8$

We denote the state space of the assembly semantics by $\mathcal{S}_{\text{asm}}$. Assembly computations are modeled by the partial function $\delta^{\text{seq}}_{\text{asm}} \in \mathcal{S}_{\text{asm}} \rightharpoonup \mathcal{S}_{\text{asm}}$. Note that the effects of exceptions like illegal page faults cannot be fully determined from the assembly-machine state. In that case, $\delta^{\text{seq}}_{\text{asm}}$ gets stuck. But with sufficient resources, a compiled C0 program does not generate exceptions during normal execution. Moreover, the memory size $s_{\text{asm}}.V$ can neither be read nor changed by the assembly machine itself but depends on the operating-system kernel. We extend the semantics accordingly in Sect. 3.

Compiler Correctness. We establish compiler correctness via a simulation relation. This relation employs an allocation function *alloc* that maps the current variables to memory locations. Based on the allocation function, we define a simulation relation $consis(alloc)(s_{\text{C0}}, s_{\text{asm}})$, which relates values of variables, pointers and the remaining program in a C0 state s_{C0} to their corresponding memory regions and the value of the program counter in an assembly state s_{asm}.

Compiler correctness states a stepwise simulation. If a C0 state simulates an assembly state, the simulation relation can again be established after a C0 step and a finite sequence of assembly steps. However, this statement is too general with respect to resource limitations and type correctness. Hence, we formulate:

Theorem 1 (Compiler Correctness). *Assuming that s_{C0} represents a well-typed C0 state, there is no runtime error in the next step, and there are sufficient resources, the following statement holds:*

$$consis(alloc)(s_{\text{C0}}, s_{\text{asm}}) \implies \exists n, alloc' : consis(alloc')(\delta^{\text{seq}}_{\text{C0}}(s_{\text{C0}}), (\delta^{\text{seq}}_{\text{asm}})^n(s_{\text{asm}}))$$

We denote f^n for the n-fold composition of a function f. Leinenbach and Petrova have formally shown this theorem in Isabelle/HOL [5].

3 Process Models

In this section, we formalize the interface between the processes and the kernel. Our formalization is based on the observation that VAMOS interacts with processes only via a well-defined interface, which is the kernel ABI. Hence, we can encapsulate processes in a self-contained input-output automaton, thereby hiding the internal state and exposing only the generic interface. We use this encapsulation in order to abstract from assembly processes as they are found in

the operational kernel models to the more abstract C0 processes. Our formalization refines the cooperative concurrent models A^{cc} from the introduction. In order to precisely model the interaction with the kernel, we refined the transition relation δ^{cc} to a partial function parametrized over an input and introduced output functions.

We define a process A_{proc} as an input-output automaton described by a tuple

$$(\mathcal{S}_{\text{proc}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \omega_{\text{proc}}, \mathit{vm\text{-}size}_{\text{proc}}, \mathit{init}_{\text{proc}}, \delta_{\text{proc}})$$

with state space $\mathcal{S}_{\text{proc}}$, input alphabet Σ_{proc}, output alphabet Ω_{proc}, output functions ω_{proc} and $\mathit{vm\text{-}size}_{\text{proc}}$, initialization function $\mathit{init}_{\text{proc}}$, and transition function δ_{proc}.

While the state space $\mathcal{S}_{\text{proc}}$ depends on the individual process model, the interface between the kernel and the processes is naturally shared by all process models. This interface is entirely defined by Σ_{proc} and Ω_{proc}.

The output alphabet Ω_{proc} enumerates all possible kernel calls. Additionally, we have to treat a few error cases. As the kernel calls are internally identified by a number, a process might specify an invalid number. This condition is represented by the special output value undefined_trap. Moreover, a process might generate exceptions like an arithmetic overflow or an illegal page fault. These exceptions are collectively represented as the value runtime_error. Finally, the output ε denotes the intention to perform a local computation.

The input alphabet Σ_{proc} reflects all kernel-initiated changes of a process. These comprise all possible responses to kernel calls, on the one hand, and the demand to change the amount of virtual memory, on the other hand. While the former are the synchronous reaction to a kernel call, the latter may be issued asynchronously at any stage of a process. In order to perform a local transition, we pass the input ε to the transition function δ_{proc}.

In order to compute the overall memory consumption of the process system, VAMOS needs to know the amount of virtual memory that is currently occupied by every process. The function $\mathit{vm\text{-}size}_{\text{proc}}$ provides the necessary information. When a new process is created, VAMOS has to transform a representation of the given binary executable file into the corresponding, initial process state. We encapsulate this transformation in the function $\mathit{init}_{\text{proc}}$.

Below, we refine this generic interface with a specific interpretation for assembly processes and C0 processes and state an extended compiler correctness theorem that relates these process models.

Assembly Processes. We reuse the state space $\mathcal{S}_{\text{asm}}$ of the assembly semantics for our assembly processes. Based on this state space, we now define the output functions ω_{asm} and $\mathit{vm\text{-}size}_{\text{asm}}$, the transition function δ_{asm}, and the initialization function $\mathit{init}_{\text{asm}}$. The function $\mathit{vm\text{-}size}_{\text{asm}}$ looks up the component V of the current state: $\mathit{vm\text{-}size}_{\text{asm}}(s_{\text{asm}}) = s_{\text{asm}}.V$. The other functions are much more complicated and we cannot fully present them here. We constrain ourselves to an exemplary excerpt.

Fig. 2 depicts the formal definition of the output function ω_{asm} and the transition function δ_{asm} for the case of a `process_clone` call. We assume that s_{asm}

$$trap(s_{\text{asm}}) \wedge simm(s_{\text{asm}}) = 2 \implies \omega_{\text{asm}}(s_{\text{asm}}) = (\mathsf{process\_clone}, s_{\text{asm}}.gpr(11))$$

$$\delta_{\text{asm}}(\mathsf{err\_unprivileged}, s_{\text{asm}}) = s_{\text{asm}} \begin{bmatrix} gpr(22) := -4 \\ pc := s_{\text{asm}}.pc + 4 \\ dpc := s_{\text{asm}}.dpc + 4 \end{bmatrix}$$

Fig. 2. Formal definition of output and transition function of assembly processes for the call `process_clone`

```
int vc_process_clone(unsigned int hn) {
  int result;
  asm {  lw(r11, r30, asm_offset(hn));
         trap(2);
         sw(r22, r30, asm_offset(result));
       };
  return result;
}
```

Fig. 3. Implementation of function `vc_process_clone` from the kernel library

is the state of an assembly process. The predicate *trap* holds iff the current instruction is a trap, and the function *simm* extracts the sign-extended immediate constant from the current instruction. If there is a trap with immediate constant 2, the output function will return the pair of process_clone and the value of register 11. Let us now assume that the kernel recognizes this output from the current process but the process is not privileged. The kernel then signals this error condition by passing the value err_unprivileged on to the current process via the transition function. In this case, the transition function updates the result register 22 with the corresponding error code and increases the program counters.

C0 Processes. Our user programs are implemented in C0. However, the pure C0 semantics cannot generate traps for the communication with the kernel. Hence, we extend the original C0 semantics with a special kernel library. This library comprises functions that use inline-assembly code to implement the kernel calls. For example, Fig. 3 shows the implementation of function `vc_process_clone`. It loads the parameter `hn` (identifying the process to clone) into register 11, performs a trap passing constant 2, and returns the value of register 22.

Recall that the sequential C0 semantics implicitly assumes sufficient memory, which is not appropriate for C0 processes. Hence, we extend the state s_{C0} of the C0 semantics by the memory size $s_{\text{C0}}.msize$ of the process and do not define the partial transition function δ_{C0} for insufficient memory. We denote the set containing all states of C0 processes by $\mathcal{S}_{\text{C0}}$. Based on this state space, we define the functions ω_{C0}, $vm\text{-}size_{\text{C0}}$, δ_{C0}, and $init_{\text{C0}}$ in analogy to their assembly-process counterparts. Function $vm\text{-}size_{\text{C0}}$ looks up the component $s_{\text{C0}}.msize$. For the other functions, we chose as prototypical example the formal definition for the call `process_clone` in Fig. 4.

$$s_{\mathrm{C0}}.prog = \text{"}e = \texttt{vc\_process\_clone}(e_0); r\text{"} \implies \omega_{\mathrm{C0}}(s_{\mathrm{C0}}) = (\mathsf{process\_clone}, \mathit{get\text{-}val}(s_{\mathrm{C0}}.mem, e_0))$$

$$s_{\mathrm{C0}}.prog = \text{"}e = \texttt{vc\_process\_clone}(e_0); r\text{"} \implies \delta_{\mathrm{C0}}(\mathsf{err\_unprivileged}, s_{\mathrm{C0}}) = s_{\mathrm{C0}} \begin{bmatrix} \mathrm{mem} := \mathit{set\text{-}val}(s_{\mathrm{C0}}.mem, e, -4) \\ \mathrm{prog} := r \end{bmatrix}$$

Fig. 4. Formal definition of the output and the transition function of C0 processes for the call `process_clone`

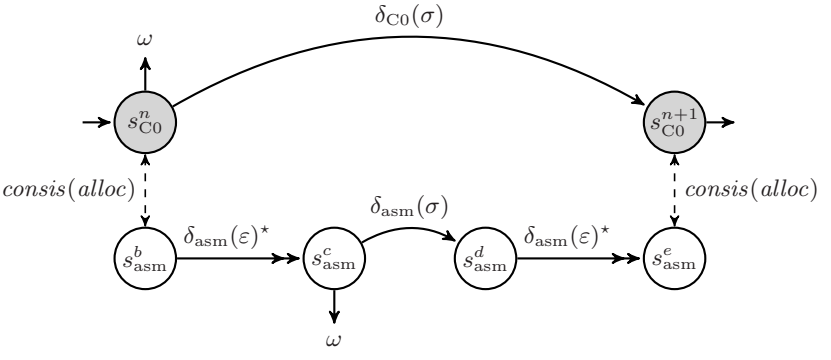


Fig. 5. Verification scheme for inline-assembly portions in the kernel library

For the C0 state s_{C0}, we consider the first statement of the program $s_{\mathrm{C0}}.prog$. If that is a function call to `vc_process_clone`, we define the output of ω_{C0} as a pair of process_clone and the function argument's value. Let us now assume that the kernel recognizes this output from the current process but the process is not privileged. The kernel then signals this error condition by passing the value err_unprivileged on to the current process via the transition function. In this case, the transition function updates the memory $s_{\mathrm{C0}}.mem$ at the address where the left-value e is stored with the corresponding error code and removes the function call from the remaining program.

Compilation. After having learned of the two process models, we now examine the compilation process and correlate both models. Our user programs are purely implemented in C0 but may contain function calls to the kernel library. For simplicity, we link on source-code level, i. e., the sources of the user program and of the kernel library are merged before compilation. Hence, the resulting C0 programs contain ordinary C0 statements on the one hand and library functions with inline assembly on the other hand.

Theorem 2 (Extended Compiler Correctness). *C0 processes simulate assembly processes.*

Proof Sketch. For ordinary C0 statements, we employ compiler correctness (Theorem 1). The correctness of the kernel library, however, can only be proven function by function on assembly level.

Fig. 5 shows the case of a library function which is called in some C0 state s^n_{C0}. From the compiler correctness theorem, we know that there exists a corresponding assembly state s^b_{asm} that satisfies the simulation relation $\mathit{consis}(\mathit{alloc})$. Now, we execute the inline-assembly code starting in one such arbitrary, fixed s^b_{asm}. When the code reaches the trap instruction in state s^c_{asm}, the assembly process has to signal the same output ω to the kernel as the C0 process does in s^n_{C0}. At this stage, the transition function δ_{asm} uses an input σ from the kernel to proceed. After further internal steps, we arrive at the end of the inline-assembly portion in a state s^e_{asm}. We compute the corresponding C0 state $s^{n+1}_{\mathrm{C0}} = \delta_{\mathrm{C0}}(\sigma, s^n_{\mathrm{C0}})$. Our library function is correctly implemented iff the pair $(s^e_{\mathrm{asm}}, s^{n+1}_{\mathrm{C0}})$ is in the simulation relation $\mathit{consis}(\mathit{alloc})$. □

A similar problem was formally proven with Isabelle/HOL by Starostin and Tsyban [6].

4 The True Concurrent Machines

In the previous section, we explained our process model. Now, we embed the processes into two true concurrent models, the VAMOS specification $A^{\mathrm{tc}}_{\mathrm{v}}$, and the *Communicating User Processes* (COUP), formally described by $A^{\mathrm{tc}}_{\mathrm{vc}}$. The former specifies the exact behaviour of our microkernel with a particular scheduler. This model is used for code verification. The latter abstracts the scheduler and focusses on the interaction of the processes with the microkernel. We need this more abstract model in order to describe the reordering of interleaved sequences.

Both models are Moore machines. We describe them with the following tuples: $A^{\mathrm{tc}}_{\mathrm{v}} = (\mathcal{S}_{\mathrm{v}}, s^0_{\mathrm{v}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\mathrm{v}}, \delta_{\mathrm{v}})$ and $A^{\mathrm{tc}}_{\mathrm{vc}} = (\mathcal{S}_{\mathrm{vc}}, s^0_{\mathrm{vc}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\mathrm{vc}}, \delta_{\mathrm{vc}})$, respectively. The state spaces $\mathcal{S}_{\mathrm{v}}$, $\mathcal{S}_{\mathrm{vc}}$ contain the initial state s^0_{v} and s^0_{vc}, respectively. For device communication, we use the input alphabet $\hat{\Sigma}$ and the output alphabet $\hat{\Omega}$. The functions ω_{v} and ω_{vc} determine the output from a current state. Finally, δ_{v} and δ_{vc} describe the transitions of the models. The notable difference between these machines regards the determinism: While $A^{\mathrm{tc}}_{\mathrm{v}}$ is fully deterministic, $A^{\mathrm{tc}}_{\mathrm{vc}}$ features a non-determinism in its scheduling decisions. Consequently, the transition relation δ_{v} is functional while δ_{vc} is not.

Below, we introduce the different components of the models side by side and present a simulation theorem with regard to an abstraction function $\mathit{abs} \in \mathcal{S}_{\mathrm{v}} \to \mathcal{S}_{\mathrm{vc}}$. Though we abstracted in $A^{\mathrm{tc}}_{\mathrm{vc}}$ from the particular scheduler policy of VAMOS, we would like to preserve the property of fairness. For an arbitrary scheduler, we cannot encode this property into the transition relation. Hence, we formulate the property over an infinite sequence of transitions and show that $A^{\mathrm{tc}}_{\mathrm{v}}$ indeed fulfils this requirement. Finally, we argue on the possibility to reorder interleaved sequences while preserving the same system behaviour.

Device Communication. Our kernel uses a memory-mapped I/O for device communication. Hence, the output alphabet $\hat{\Omega}$ comprises read and write accesses to device addresses. The input alphabet $\hat{\Sigma}$ consists of interrupt lines and optionally incoming data. Hillebrand *et al.* [7] have described our device interface in detail.

State Spaces. A state $s_\mathrm{v} \in \mathcal{S}_\mathrm{v}$ comprises the following components:

- The partial function $s_\mathrm{v}.\mathit{procs}$ maps the process identifiers (PIDs) of the currently active processes to their assembly states $s_\mathrm{asm} \in \mathcal{S}_\mathrm{asm}$. For inactive processes, this function is undefined.
- Priorities are assigned to each PID of the active processes with the partial function $s_\mathrm{v}.\mathit{priodb}$.
- All other scheduling information is kept in the component $s_\mathrm{v}.\mathit{schedds}$.
- The partial function $s_\mathrm{v}.\mathit{rightsdb}$ maps PIDs to a data structure for the management of IPC rights and the set of privileged processes.
- Finally, the component $s_\mathrm{v}.\mathit{devds}$ contains data for device communication.

The corresponding state $s_\mathrm{vc} = \mathit{abs}(s_\mathrm{v})$ inherits all components of s_v except for $s_\mathrm{v}.\mathit{schedds}$, which is replaced by a current-process indicator. We retain the current process in s_vc in order to compute the output from the current state. The output function ω_vc signals the demand for device communication. In order to determine this demand, we need to employ the output function ω_asm of the current process. Consequently, we fix this process beforehand instead of including transitions for all ready processes in the transition relation.

We take a closer look at the scheduling data structures because they concern us in the following sections. The component $s_\mathrm{v}.\mathit{schedds}$ itself is subdivided into components. The current time $\mathit{time} \in \mathbb{N}$ is a counter for the clock ticks. Process-specific scheduling information for active processes is collected in the partial function *procdb* that maps PIDs to a record of (a) the time slice *tsl*, (b) the amount of consumed time *ctsl*, and (c) the absolute timeout *to*. If a process is found to be computing when a timer interrupt raises, the component *ctsl* is increased until the process has finally run for *tsl* ticks. In this case, another process is scheduled. If a process calls the kernel for IPC and no partner is ready for communication, the absolute timeout *to* is computed from the current time and the relative timeout that is specified with the call.

Moreover, the scheduler maintains different queues for scheduling. They are represented as finite sequences in A_v^tc. Namely, there is a ready queue $\mathit{ready}(\mathit{prio})$ of schedulable processes for each priority $\mathit{prio} \in \{0, 1, 2\}$. The processes that cannot currently be scheduled (because they are waiting for an IPC partner) are held in a queue named *wait*.

In Vamos, the current process is the first process in the highest, non-empty ready queue. If all ready queues are empty, the current process is undefined. Formally, we define the function *cup* as:

$$\begin{aligned}\mathit{cup}(s_\mathrm{v}.\mathit{schedds}) = p \iff\ & \exists i : s_\mathrm{v}.\mathit{schedds}.\mathit{ready}(i) = (p, \dots) \wedge \\ & \forall\, j > i : s_\mathrm{v}.\mathit{schedds}.\mathit{ready}(j) = ()\end{aligned}$$

Transitions. A transition $\delta_{\mathrm{v}}(\hat{\sigma}, s_{\mathrm{v}})$ under the device input $\hat{\sigma} \in \hat{\Sigma}$ has up to three phases:

1. If the current process $cp = cup(s_{\mathrm{v}}.schedds)$ is defined, we consult its output $\omega_{\mathrm{asm}}(s_{\mathrm{v}}.procs(cp))$ and compute the response according to the current VAMOS state. For instance, if a process calls `process_clone`, we check for sufficient privileges and resources and choose the corresponding response $\sigma \in \Sigma_{\mathrm{proc}}$ for success or failure. With this response, we advance the current process: $s_{\mathrm{v}}\,[procs(cp) := \delta_{\mathrm{asm}}(\sigma, s_{\mathrm{v}}.procs(cp))]$.
2. If the timer-interrupt line is raised, the scheduler increases the clock-tick counter $s_{\mathrm{v}}.schedds.time$ and the consumed time $s_{\mathrm{v}}.schedds.procdb(cp).ctsl$ of the current process. Moreover, the scheduler wakes up all processes p with elapsed timeouts, i. e., where the absolute timeout $s_{\mathrm{v}}.schedds.procdb(p).to$ is less than or equal to the current time $s_{\mathrm{v}}.schedds.time$.
3. Finally, VAMOS delivers interrupts to waiting drivers and saves the remaining interrupts for later delivery in $s_{\mathrm{v}}.devds$.

The transitions δ_{vc} behave very similarly. However, a transition $s_{\mathrm{vc}} \xleftrightarrow{\sigma} s'_{\mathrm{vc}} \in \delta_{\mathrm{vc}}$, obtains cp directly from $s_{\mathrm{vc}}.cup$. Moreover, the only visible effect of the second phase is the wake-up of certain waiting processes. This effect is simulated non-deterministically and independently from the timer interrupt.

Simulation Theorem. We would like to show that our more abstract model $A^{\mathrm{tc}}_{\mathrm{vc}}$ simulates the kernel specification $A^{\mathrm{tc}}_{\mathrm{v}}$. We formulate this fact over an infinite input sequence $inputs \in \mathbb{N} \to \hat{\Sigma}$ that maps a step number to a particular device input.

Theorem 3 (Equivalence of Transition Sequences). *The initial states are equivalent, i. e., $abs(s^0_{\mathrm{v}}) = s^0_{\mathrm{vc}}$, and their equivalence is preserved for all device inputs σ after each transition, i. e.,*

$$abs(s_{\mathrm{v}}) \xleftrightarrow{\sigma} abs(\delta_{\mathrm{v}}(\sigma, s_{\mathrm{v}}))$$

Proof Insights. We proved this statement formally in Isabelle/HOL [5]. As we could just glance at the transition relations in this paper, we can only summarize a few insights from our proof. All in all, the verification was straightforward because we reused the infrastructure of $A^{\mathrm{tc}}_{\mathrm{v}}$ to a large extent in $A^{\mathrm{tc}}_{\mathrm{vc}}$. The biggest difficulty we faced was in the verification of IPC because of a considerably simpler modelling in $A^{\mathrm{tc}}_{\mathrm{vc}}$, which became possible after the scheduler was abstracted. □

Fairness. We formulate fairness over infinite transition sequences, which we represent as two functions *states* and *inputs* that map the step number to the state and the input, respectively. We assume a live timer device, i. e., $\forall n\, \exists m \geq n : is\_timer\_on(inputs(m))$. A process system with a single priority behaves fair if an arbitrary, fixed process *pid* eventually (a) is not active, (b) has a changed priority, (c) starves in an IPC operation with an infinite timeout, or (d) advances. This property also holds in VAMOS for a process system with multiple priorities if several restrictions apply for the processes with lower priority. Though some

of these restrictions could be removed by a more sophisticated implementation, we constrain ourselves to a typical implementation. That means that fairness is only preserved if the current maximum priority is infinitely often low at those points in the transition sequence, where the timer interrupt is raised.

Formally, we define the liveness property:

$$
\begin{aligned}
&\mathit{isFair}(\mathit{inputs}, \mathit{states}) \equiv \forall \mathit{pid}\ k : \\
&\quad \forall n \geq k : \exists m \geq n : \mathit{has\_maxprio}(\mathit{states}(m), \mathit{pid}) \wedge \mathit{is\_timer\_on}(\mathit{inputs}(m)) \\
&\quad \Longrightarrow\ \exists l \geq k : \neg \mathit{defined}(\mathit{states}(l).\mathit{procs}(\mathit{pid})) \vee \\
&\qquad\qquad\qquad \mathit{states}(l+1).\mathit{priodb}(\mathit{pid}) \neq \mathit{states}(l).\mathit{priodb}(\mathit{pid}) \vee \\
&\qquad\qquad\qquad \mathit{starving\_infinite\_ipc}(\mathit{states}, l, \mathit{pid}) \vee \\
&\qquad\qquad\qquad \mathit{progress}\,(\mathit{states}(l).\mathit{procs}(\mathit{pid}), \mathit{states}(l+1).\mathit{procs}(\mathit{pid}))
\end{aligned}
$$

If a process *pid* has the currently maximal priority at recurring observation points m with $\mathit{is\_timer\_on}(\mathit{inputs}(m))$, one of the conditions (a)–(d) eventually holds. Formally, these conditions are expressed as (a) the function *procs* of the current state is undefined for *pid*, (b) inequality of the priorities in consecutive states, (c) the predicate $\mathit{starving\_infinite\_ipc}(\mathit{states}, l, p)$, which examines a sequence of states *states*, holds iff the process p is pending in an IPC operation with an infinite timeout since step l, (d) the predicate $\mathit{progress}(p, q)$, which holds iff q can be produced by one or more transitions starting at p.

Theorem 4 (Fairness). *The* VAMOS *scheduler is fair, i. e., with the set* $\mathcal{R}_\mathrm{v}$ *containing the infinite transition sequences of* A_v^tc*, we can state*

$$\forall\,(\mathit{states}, \mathit{inputs}) \in \mathcal{R}_\mathrm{v} : \mathit{isFair}(\mathit{inputs}, \mathit{abs} \circ \mathit{states})$$

Proof Sketch. An arbitrary, fixed process *pid* can either be inactive, waiting, or ready. In the first case, Condition (a) of the predicate *isFair* holds immediately. If a process is waiting, it has called the kernel for an IPC operation and no partner is ready for it. In this case, there will eventually be a partner issuing a kernel call for IPC, the operation will time out, or the process is starving in an infinite IPC. The latter case corresponds to Condition (c), the former two imply progress (Condition (d)) because the kernel returns a result to the process.

Finally, there remains the case where the process is currently ready. Then, the process resides in the ready queue that corresponds to its priority. The process will be dequeued only if it becomes inactive, or if its priority changes (Conditions (a) and (b)). Assuming that the process has the current maximum priority infinitely often while the timer interrupt is active, the process will move forward in the ready queue until it is the first one. We know this fact because the scheduler will charge the current process if the timer interrupt is active and timeslices are bound by a fixed value. Hence, the process *pid* will eventually be the first in its ready queue and thus, when it eventually has the maximum priority, it is the current process.

Usually, the current process advances immediately. Most notably, our implementation guarantees liveness, i. e., a started user process performs at least one step between two subsequent kernel entries. Still, there might be no immediate

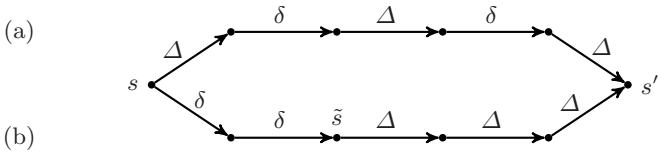


Fig. 6. Reordering transitions of the global system

progress if the current process calls the kernel for an IPC operation. In this case, however, the kernel will enqueue the process in the wait queue, and we have shown fairness for all processes in this queue.

This proof has been formally developed in Isabelle/HOL [5]. □

Shifting Scheduling Decisions. We have introduced $A^{\mathrm{tc}}_{\mathrm{vc}}$ as a means to reinterpret execution traces of the $A^{\mathrm{tc}}_{\mathrm{v}}$ machine such that rescheduling only takes place when the current process s_{asm} is about to execute a kernel primitive, i. e., $\omega_{\mathrm{asm}}(s_{\mathrm{asm}}) \neq \varepsilon$. This reinterpretation is possible if a process cannot observe that it was interrupted and resumed except at a kernel call. If processes do not observe scheduling decisions, the whole process system behaves confluently. We formulate:

Theorem 5 (Reordering). *Scheduling decisions of a process pid are confluent between two adjacent kernel calls of this process.*

Proof Sketch. For this proof, we can employ trace theory. In short, transitions can be commuted if they are independent of each other. Fig. 6 shows (a) an interleaved sequence of transitions between two states s and s' of the global system together with (b) a reordered sequence with a cooperative concurrent segment between s and $\tilde{s}$. In the figure, we have marked all transitions that do not affect the process *pid* by Δ and the local assembly steps of the process *pid* by δ. These transitions do not interfere with each other and can thus be commuted.

Non-commutable transitions, called *synchronization points*, affect simultaneously the kernel data structures and the considered process *pid*. The most prominent representative is a kernel call of *pid*. However, this is not the only one: The kernel may asynchronously change the memory size of processes. Strictly speaking, these transitions break our non-interference assumption and thus inhibit confluent reordering. However, we know from the kernel specification that the memory size is only changed if one of the privileged processes requests it. In practice, there should be a protocol between the processes ensuring that a process is always on a synchronization point if its memory is changed. Hence, we can establish the reordering theorem if and only if the privileged processes request to change the memory size of process *pid* only at synchronization points.

As mentioned earlier, the set of privileged processes is usually very small. In a system with a single privileged process, the memory size of this process cannot be changed asynchronously. Hence, we may reason about this process in a cooperative concurrent fashion and show that it will not change the size of other processes unless they have requested it via IPC – and hence have reached

a synchronization point. With this knowledge, we may reason about all other processes in a cooperative concurrent fashion. □

5 Conclusion and Future Work

Related Work. Bevier [8] was a precursor in operating-system verification. However, he verified a fairly simple kernel in comparison to modern microkernels.

Many recent projects undertake verification efforts on modern microkernels. Among them are VFiasco [9] and its successor Robin [10], L4.verified [11], EROS [12] and its successor Coyotos [13], as well as the FLINT project [14]. Though these projects may have achieved some advances, they all focus on microkernel verification but do not ascend towards the verification of user programs.

Recently, Hobor *et al.* [15] have proposed a verification method for concurrent programming languages. The group assumes a fairly abstract language model and extends it with threads that communicate using locks while competing for a shared memory. Besides the differences in the communication model, this work is complementary to ours because Hobor *et al.* just assume a cooperative concurrent environment.

Achievements. Though kernel verification is often motivated as a foundation, the verification of operating-system components running on a kernel remains an open problem. So far, existing approaches only *postulate* a cooperative concurrent environment with fair scheduling.

In this paper, we bridge the gap between these cooperative concurrent models and a realistic, true concurrent execution machine. Thus, we provided the foundation for the verification of concurrent C0 programs. In particular, we extended a sequential C0 semantics and a sequential assembly semantics with kernel primitives, extended an existing compiler-correctness theorem for the sequential semantics accordingly, formally proved that our microkernel is fair, and showed under which circumstances execution traces in the true concurrent system can be reordered to cooperative concurrent ones. We have formally developed the proofs for Theorems 3 and 4 in Isabelle/HOL within ten person months.

Future Work. Besides the formalization of Theorems 2 and 5 in Isabelle/HOL and the technical integration of the kernel primitives into Simpl, we see the following important extension of our work: So far, reasoning in terms of process-local C0 programs is inherently only possible for partial correctness. In a process-local view of communication primitives like `ipc_receive`, it can not always be inferred if a process can actually continue execution or will get stuck because there is no communication partner. In order to establish total correctness for processes (what the Simpl framework potentially can), it is necessary to consider the global system state in a suitably extended Hoare-Logic Simpl* allowing us to reason about the system of processes as such.

Acknowledgements. We thank Sarah Hoffmann, Norbert Schirmer, and Irena Dotcheva for reviewing, constructive criticism and helpful suggestions.

References

1. Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Systems Software Verification. Elsevier, Amsterdam (to appear, 2008)
2. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, TU Munich (2006)
3. Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM, pp. 2–12. IEEE Computer Society, Los Alamitos (2005)
4. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. STTT 8(4-5), 411–430 (2006)
5. Verisoft Project: Verisoft repository (2008), http://www.verisoft.de/VerisoftRepository.html
6. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Systems Software Verification. Elsevier, Amsterdam (to appear, 2008)
7. Hillebrand, M.A., In der Rieden, T., Paul, W.J.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD, pp. 309–316. IEEE, Los Alamitos (2005)
8. Bevier, W.R.: Kit and the short stack. J. Autom. Reasoning 5(4), 519–530 (1989)
9. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: the VFiasco project. In: ACM SIGOPS European Workshop, pp. 165–169. ACM, New York (2002)
10. Tews, H.: Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. In: C/C++ Verification Workshop, technical report ICIS–R07015, Radboud University Nijmegen, pp. 59–68 (2007)
11. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: taking microkernels to the next level. Operating Systems Review 41(4), 3–11 (2007)
12. Shapiro, J.S., Weber, S.: Verifying the EROS confinement mechanism. In: IEEE Symposium on Security and Privacy, pp. 166–176 (2000)
13. Shapiro, J., Doerrie, M.S., Northup, E., Sridhar, S., Miller, M.: Towards a verified, general-purpose operating system kernel. In: FM Workshop on OS Verification. Technical Report 0401005T-1, National ICT Australia, pp. 1–19 (2004)
14. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 189–206. Springer, Heidelberg (2007)
15. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)

Boogie Meets Regions: A Verification Experience Report

Anindya Banerjee[1,⋆], Mike Barnett[2], and David A. Naumann[3,⋆⋆]

[1] Kansas State University, Manhattan KS 66506, USA
[2] Microsoft Research, Redmond WA 98052, USA
[3] Stevens Institute of Technology, Hoboken NJ 07030, USA

Abstract. We use region logic specifications to verify several programs exhbiting the classic hard problem for object-oriented systems: the framing of heap updates. We use BoogiePL and its associated SMT solver, Z3, to prove both implementations and client code.

1 Introduction

Many programs use dynamically allocated mutable storage, which poses challenges for encapsulation and exacerbates the frame problem: how to specify that "everything else is unchanged" for mutable state not directly named by program variables? Several lines of work approach this problem in terms of *footprints*, i.e., the sets of locations that are written or read by a phrase of a program or specification.

- In the Boogie methodology [5], "owned state" is represented by a mutable ghost field that points to an object's owner if it has one. An *effect specification* ("modifies" clause) that licenses update of an object o implicitly licenses update of the objects transitively owned by o.
- In Separation Logic [21], if P is the precondition of a procedure then the procedure may be viewed as owning the state on which P depends. It has license to modify that part of the heap but no other.
- Kassios [17] shows how to manipulate footprints explicitly, in ghost variables, rather than implicitly (in formulas) or indirectly (via transitive ownership).

A key point is that if the read footprint of a formula or pure expression is disjoint from the write footprint of a command, then the command preserves the value of the formula or expression.

Kassios works in a higher order logic, which can directly express that a formula depends on certain locations. He shows how reasoning idioms developed in Boogie or separation logic can be elegantly and effectively articulated using explicit footprints. His approach does require a single discipline to be imposed on all parts of all programs. He develops a relational refinement calculus, following Hehner [15].

⋆ Partially supported by US NSF awards CNS-0627748 and ITR-0326577 and by a sabbatical visit at Microsoft Research, Redmond.
⋆⋆ Support from NSF (CNS-0627338, CRI-0708330, CCF-0429894) and Microsoft Research.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 177–191, 2008.

Smans et al [24] explore Kassios' approach in terms of conventional sequential specifications with distinct first order precondition, postcondition, and effect. They focus on framing of pure methods used in specifications for data abstraction. They report on encouraging case studies using a prototype verifier based on an SMT solver.

Banerjee et al [4] also explore the approach using first-order specifications, focusing on foundational justification in terms of a Hoare logic that uses footprint expressions in the effects clause. Their Region Logic features a frame rule inspired by that of separation logic (which in turn adapts Hoare's rule of Invariance) to encompass interference via mutable heap objects. In region logic, the frame rule involves a static analysis for read effects of formulas. The effect **rd** $G.f$ expresses that field f of some objects in G may be read. A region expression G denotes a set of non-null object references and footprints are of the form $G.f$. This treatment caters for notations like $G.f \subseteq G'$ which says that the f-image of G is a subset of G'.[1] Region logic features the use of region fields and variables to encode dynamic frames whereas Smans *et al.* use pure methods. Banerjee *et al.* claim that: "A benefit of treating regions as ghost state is that it can be done using first-order specification languages based on classical logic with modest use of set theory. Thus it fits with mostly-automated tools based on verification condition generation. . . ."

This paper reports on some case studies to investigate Banerjee *et al.*'s claim. We demonstrate the utility of using region specifications for

- Local reasoning about data structures (Sect. 4),
- Encapsulation, even in the presence of callbacks (Sect. 5),
- Layered abstractions, using the examples from Smans et al. [24] but without the need for conditional effects (Sect. 6).

In addition, our experience supports their suggestion that the region logic provides a neutral framework in which disciplines such as ownership can be, but need not be imposed system-wide.

In ongoing work we are investigating decision procedures for region logic primitives such as $G.f \subseteq G'$. Here, we use the Z3 solver [12] which does not have a decision procedure for set theory. This poses another research question: Is a first order axiomatization of set theory effective for automating verification of programs and specifications that encode footprints in ghost state? Our answer is yes (Sect. 9) and no (Sect. 7).

Rather than implement a verification condition generator from scratch for our assertion language, we use the BoogiePL intermediate language advocated at VSTTE'05 by Barnett et al [6]. The BoogiePL tool [13] generates verification conditions suited for SMT solvers such as Simplify [14], the CVC family [8], and Z3 [12]. These integrate multiple decision procedures with heuristic instantiation of quantifiers driven by pragmas called *triggers*. They have been used with some success in verifiers like ESC/Java and Spec#, where user interaction is limited to the insertion of assert and assume statements and loop invariants. Our experiments consist entirely of manually written encodings in BoogiePL of the example programs and specifications, with Z3 as solver. We are therefore in a position to provide triggers on an ad hoc basis.

[1] These features are useful for reasoning about simulations, as in the context of information flow [1,2].

2 Relevant Features of BoogiePL

A BoogiePL [13] program is a collection of procedure specifications and implementations, together with global variables, uninterpreted functions, and axioms. The Boogie tool generates verification conditions for the procedure implementations with respect to their specifications. These conditions, together with the axioms, are translated into an input format for SMT solvers, in our case Z3. We do not explain that translation: the verification conditions represent the weakest precondition of the program [7]. Instead we focus on the features provided in BoogiePL and their use for encoding region specifications and programs. Aside from regions, our encodings are similar to those automatically generated by Spec#, but we omit the treatment of source level data types.

Types. The type system of BoogiePL is coarser than that of a high-level object-oriented language: objects are represented by the type **ref** (of which **null** is a distinguished element). The other primitive types are **bool**, **int**, and **name**, all subtypes of **any**. We use the type **name** for fields. Another subtype of **any** is a type constructor for maps, using array-style notation. In particular, we encode regions as [**ref**]**bool**, denoting a boolean-valued function on references.

Heaps. The heap is modeled as an infinite two-dimensional array, roughly of type [**ref**, **name**]**any**, mapping pairs of object references and field names to values. Logically, all objects "have" all fields, but of course, each object uses only those fields which are defined in the corresponding high-level program. There is no built-in notion of an object being allocated or not. We model it using an explicit boolean field, $alloc$, and assume that the field is set appropriately in the necessary places (see below).

We declare a single global variable, "$Heap$". All operations involving the heap are explicit in our code and specifications: What would be written $o.f$ in source code, and in region logic specifications, is here written as $Heap[o, f]$. We write "Hp" for the type of heaps. (It is not actually [**ref**, **name**]**any** but rather a refinement that uses a form of type dependency to distinguish integer fields from reference fields but we elide the details [13].) We write **field** for the type of fields, again hiding the dependency notation for field names.

Source programs only compute heaps h that are self-contained in the sense that for any allocated object o, and for any field f of o, if f's content is a reference, p, then p is also allocated in h. We use an uninterpreted predicate, *GH*, read "is good heap", and in our initial experiments an axiom was used saying that $GH(h)$ implies self-containment. It was not a defining axiom, as we expected for some purposes one would want additional conditions such as typing. It turned out that our results do not even need self-containment. There is also a predicate *GO* ("is good object") with a defining axiom that says $GO(o, h)$ iff $GH(h)$ and o is non-null and o is allocated (i.e., $h[o, alloc]$ is true).

Procedures and functions. Procedures in BoogiePL model the specification of imperative code (with optional implementations). Functions introduce arbitrary axiomatizations and model user-defined methods within specifications. All parameters are explicit, including the receiver of an instance method and the return value.

Procedures have *preconditions* and *postconditions* for encoding the assertions they demand of a caller and the guarantees that a caller can assume, respectively. *Free* preconditions and postconditions are not checked (asserted) but instead assumed. Such

assumptions model facts that are guaranteed by the high-level programming language. Free preconditions represent facts that a callee can depend on, while free postconditions facts that a caller can depend on. For example, we use a free precondition that the receiver of an instance method is non-null and allocated and a free postcondition that all allocated objects remain so (predicate $NoDeallocs$). In this paper we elide most of the free conditions.

Implementations are written using the usual high-level control structures of assignment statements, conditional statements, and while loops. Loops are allowed to have loop invariants. There are also assert statements, which introduce a proof obligation, and assume statments which introduce unchecked facts. For example, every assignment to the heap is followed by an assumption $GH(Heap)$ which is justified because the source language would make *GH* an all-states invariant. Finally, the statement **havoc** o loses all information about the value of the variable o.

When calls of (source code) methods are allowed in specifications, those methods must be *pure*, i.e., not modify any pre-existing state. All calls to a pure method, m, in specifications are encoded by an uninterpreted function, $\#m$. The axioms defining $\#m$ are either derived from the specification for m [11] or else directly from the implementation of m [24].

Allocation. We encode the high-level source statement $o := \textbf{new}\ T(\ldots)$ as

$$\textbf{havoc}\ o;\ \textbf{assume}\ \neg Heap[o, alloc];\ \textbf{call}\ T..ctor(\ldots);$$

where $T..ctor$ is the procedure that encodes the appropriate constructor for the type T. For brevity we use the high-level statement in this paper. All constructors have a free postcondition that the object is a good object, hence allocated. That is, there is no explicit state change that updates the $alloc$ field to allocate an object, but the effect of calling the constructor provides the same functionality. Constructors also have a free precondition that the object is allocated. The free precondition that o is allocated does not lead to a contradiction with the assumption $\neg Heap[o, alloc]$ because only *checked* preconditions become proof obligations (assertions) at call sites to the constructor. The result is that code within a constructor can use the object, e.g., as a parameter to a call.

The default write effect of any constructor is that only fields of the object under construction can be updated.

Quantifiers. Quantifiers are written in BoogiePL as $(Q\ v \bullet\ body)$ where Q is the kind of the quantifier, v is a list of bound variables and their types and $body$ is a predicate dependent on the variables in v. For axioms involving quantifiers, immediately following the $\bullet$ and before the quantifier body a *trigger* might appear. Triggers are syntactic patterns delimited by $\{...\}$ and are used to provide hints to the SMT prover on how to instantiate the quantified variable. Triggers are discussed in more detail in Sect. 7.

Modifies clauses. BoogiePL specifications include a *modifies* clause, but this merely lists variables. Thus every mutator method lists $Heap$, and for brevity in this paper we elide that. More important are the source level effect specifications. The high level specification of a method might include effect $\textbf{wr}\ \langle o \rangle.f$, for a parameter o and field f

(this region logic notation corresponds to just $o.f$ in the modifies clause of Spec# or JML). As usual, this translates to a postcondition of the form

$$\forall p : \mathbf{ref}, g : \mathbf{field} \bullet GO(p, \mathbf{old}(Heap)) \implies \mathbf{old}(Heap)[p, g] = Heap[p, g] \vee (p = o \wedge g = f) \vee \ldots \quad (1)$$

where the elided part comes from other effect specifications. There is no restriction on modifying any field of an object that is allocated during the method call.

3 Regions

Recall that regions, i.e., reference sets, are encoded by their characteristic functions of type [**ref**]**bool**. Region logic features effects of the form **wr** $this.O\bullet f$ where O is a region type field and f a fieldname. The effect **wr** $this.O\bullet f$ is translated exactly as Equation 1, but replacing $p = o$ with the test whether p is member of $\mathbf{old}(Heap)[this, O]$. Note that write effects are interpreted in the initial state. In our experiments, the only form of region expression used are field dereferences and variables. Region fields and variables serve only for reasoning, but BoogiePL does not distinguish ghost state from program state. (Ghosts would be marked as such in a source level notation like Spec#.)

Although sets are represented by the interpreted type [**ref**]**bool**, this BoogiePL type comes equipped only with equality test and the select/update operations. Other set theoretic operations and predicates can be given straightforward defining axioms. Here are some sample declarations.

function $SingletonSet(\mathbf{ref})$ **returns** ([**ref**] **bool**);
axiom $\forall r : \mathbf{ref}, o : \mathbf{ref} \bullet \{SingletonSet(r)[o]\}\ SingletonSet(r)[o] \iff r = o$;

function $DisjointSet([\mathbf{ref}]\ \mathbf{bool}, [\mathbf{ref}]\ \mathbf{bool})$ **returns** (**bool**);
axiom $\forall a : [\mathbf{ref}]\ \mathbf{bool}, b : [\mathbf{ref}]\ \mathbf{bool} \bullet \{DisjointSet(a, b)\}$
$DisjointSet(a, b) \iff \forall o : \mathbf{ref} \bullet \{a[o]\}\{b[o]\}\ \neg a[o] \vee \neg b[o]$;

In the $DisjointSet$ axiom, either $a[o]$ or $b[o]$ can be triggers for the inner quantifier. In the sequel, we write $\varnothing$ for $EmptySet$, $\{o\}$ for $SingletonSet(o)$, $A \cup B$ for $UnionSet(A, B)$, $A = B$ for $EqualSet(A, B)$ to save space. To avoid clutter in this paper, we omit triggers from most axioms.

Predicate $GR(G, h)$ is defined to say that every object in region G is a good object in h. As mentioned in Section 1, region logic features predicates involving the f-image of a region. For example, region G is closed with respect to field f (of type **ref**) in heap h provided that for every object o in G, the value of $o.f$ is either **null** or in G:

function $RegionClosed(G : [\mathbf{ref}]\mathbf{bool}, h : Hp, f : \mathbf{field})$ **returns** (**bool**);
axiom $\forall G : [\mathbf{ref}]\mathbf{bool}, h : Hp, f : \mathbf{field} \bullet$
$RegionClosed(G, h, f) \iff \forall o : \mathbf{ref} \bullet G[o] \implies h[o, f] = \mathbf{null} \vee G[h[o, f]]$

Region G is *fresh* with respect to an initial heap provided its elements were unallocated:

function $fresh(h : Hp, k : Hp, G : [\mathbf{ref}]\mathbf{bool})$ **returns** (**bool**);
axiom $\forall h : Hp, k : Hp, G : [\mathbf{ref}]\mathbf{bool} \bullet$
$fresh(h, k, G) \iff GH(h) \wedge GR(G, k) \wedge AllNewRegion(h, G)$

The predicate $AllNewRegion(h, G)$ asserts that all objects in region G are unallocated in heap h. The predicate $DifferenceIsNew(h, G, G')$ holds provided G' is a newly allocated region with respect to both G and h: in other words if o is any object allocated in h, either o is **null** or o is an element of G or o is not an element of G'.

4 Local Reasoning Example: List Copy

Fig. 1 shows the body of a $ListCopy$ method that takes a linked list of nodes as input and produces as output a new list which is a copy of the original list. Our goal is to verify that if the original list is non-empty and resides in a region —say $origRg$— then the copied list is also non-empty and resides in a region, say, $newRg$, that is fresh and disjoint from $origRg$. (We stress that we are not verifying full functional correctness, namely, that the new list is indeed a copy of the original.) For this purpose, $origRg$ and $newRg$ would be auxiliary variables, scoped over both the requires and ensures clause. Such variables are supported by JML and Region Logic but not the current version of BoogiePL, so for illustration we make them local variables instead of parameters and use assert and assume statements to encode the specification.

procedure $ListCopy(root : \mathbf{ref})$ **returns** $(result : \mathbf{ref})$
var $newRoot, prev, tmp, oldListPtr : \mathbf{ref};\ newRg, origRg : [\mathbf{ref}]\mathbf{bool}$
$\{$ **assume** $root = \mathbf{null} \vee (origRg[root] \wedge RegionClosed(origRg, Heap, nxt));$
$newRoot := \mathbf{null};\ tmp := \mathbf{null};\ oldListPtr := root;\ newRg := \varnothing;$
if $(oldListPtr \neq \mathbf{null})\{$
$newRoot := \mathbf{new}\ Node();\ newRg := newRg \cup \{newRoot\};$
$prev := newRoot;\ oldListPtr := Heap[oldListPtr, nxt];$
while $(oldListPtr \neq \mathbf{null})\{$
$tmp := \mathbf{new}\ Node();\ newRg := newRg \cup \{tmp\};$
$Heap[prev, nxt] := tmp;$ **assume** $GH(Heap);$
$prev := tmp;\ oldListPtr := Heap[oldListPtr, nxt];\}\}$
$result := newRoot;$
assert $(root = \mathbf{null} \wedge result = \mathbf{null}) \vee (root \neq \mathbf{null} \wedge RegionClosed(newRg, Heap, nxt)$
$\wedge\, newRg[result] \wedge fresh(\mathbf{old}(Heap), Heap, newRg) \wedge DisjointSet(newRg, origRg));\}$

Fig. 1. List Copy implementation with embedded specifications (see text)

At the beginning of every loop iteration, the variable $oldListPtr$ contains a pointer to the remainder of the original list that is yet to be copied. The variable $prev$ contains a pointer to the current end node of the new list: the new list grows when a new node pointed to by tmp is added at its end. All nodes in the copied list exist in $newRg$. This region is fresh with respect to the heap at the entry to the method. Freshness of each node in $newRg$ is ensured by the allocation of a new object (Sect. 2). The constructor call for $Node$ ensures that tmp is a good object, i.e., non-null and allocated. Since tmp is assumed to be unallocated before the constructor call, $\{tmp\}$ is a fresh region.

The loop invariant below is obtained from the postconditions by standard heuristics, except for the last conjunct which should be mechanically generated for every loop

translated from a high-level source program; ghost variable $loopPreHeap$ is the snapshot of the heap before the loop body is entered.

$$newRg[newRoot] \wedge RegionClosed(newRg, Heap, nxt)$$
$$\wedge fresh(\mathbf{old}(Heap), Heap, newRg) \wedge NoDeallocs(loopPreHeap, Heap)$$

This suffices to prove $DisjointSet(newRg, origRg)$.

5 Encapsulation Example: Subject/Observer

Fig. 2 shows a more involved example, *Subject/Observer*. A subject, s, has an internal state in field val and a pointer to a list of observers with typical element, o. The head of the list is reached via $s.obs$ and other observers in the list are reached following the observers' nxt fields. The entire list of observers resides in a region contained in (ghost) field O of s. The observer o maintains a pointer to its subject in its sub field and $o.cache$ contains o's current view of s's internal state.

Method $register$ adds an object o to the region O of a subject, s, and notifies o of s's current state. When the subject's state is $update$d, it notifies all of its observers. The purpose of method $notify$ is to update an observer's view of its subject's internal state. Note that $notify$ is called on an observer from within the $update$ method and this results in a callback to the $Subject$'s get method via $this.sub.get()$.

For the specifications of the methods of Fig. 2, the predicates $SubObs$, Sub, Obs are used (inspired by Parkinson [22]). The predicate $Sub(s, v)$ says that the current internal state of subject s is v and all observers of s are in a list which lies in region $s.O$. The predicate $Obs(o, s, v)$ says that o is an observer of subject s and that v is o's view of s's internal state. Finally, $SubObs$ is a predicate for the entire aggregate structure comprising an instance of $Subject$ together with its $Observer$s. $SubObs(s, v)$ holds for a subject s with internal state v when $Sub(s, v)$ holds and for each observer o in s's list of observers, $Obs(o, s, v)$ holds.

```
// Observer: fields sub, nxt : ref, cache : int
// methods: ctor (constructor), notify
procedure notify(this : ref){
  var tmp : int;
  tmp := call get(Heap[this, sub]);
  Heap[this, cache] := tmp; assume GH(Heap);}

procedure ctor(this : ref, s : ref){
  Heap[this, sub] := s;
  assume GH(Heap);
  Heap[this, nxt] := null;
  assume GH(Heap);
  call register(s, this);}

// Subject: fields obs : ref, val : int, O : [ref]bool
// methods: ctor (constructor), register, add, update, get
procedure update(this : ref, n : int){
  var o : ref; var r1 : [ref]bool;
  Heap[this, val] := n; assume GH(Heap); o := Heap[this, obs]; r1 := ∅;
  while (o ≠ null){call notify(o); r1 := r1 ∪ {o}; o := Heap[o, nxt];}
```

Fig. 2. Subject/Observer implementation excerpts

$$\textbf{function } List(o:\textbf{ref}, h:Hp) \textbf{ returns } ([\textbf{ref}]\textbf{bool});$$
$$\textbf{axiom } \forall o:\textbf{ref}, h:Hp \bullet o = \textbf{null} \iff List(o,h) = \varnothing$$
$$\textbf{axiom } \forall o:\textbf{ref}, h:Hp, r:[\textbf{ref}]\textbf{bool} \bullet o \neq \textbf{null} \implies List(o,h) = \{o\} \cup List(h[o,nxt],h)$$
$$\textbf{axiom } \forall h:Hp, k:Hp, o:\textbf{ref} \bullet List(o,h) = List(o,k) \impliedby (\forall p:\textbf{ref} \bullet p \neq \textbf{null} \wedge List(o,h)[p] \wedge List(o,k)[p] \implies h[p,nxt] = k[p,nxt])$$

Fig. 3. List axioms

Unlike Parkinson's formulation in separation logic, our version includes the heap as explicit parameter, but that would be hidden in source level syntax. In Fig. 3 we introduce a function $List$ so that $List(o,h)$ is a set containing all objects reachable from o following nxt. Using $List$, the defining axioms for Sub, Obs and $SubObs$ look as follows:

$$Sub(s,v,h) \iff GO(s,h) \wedge h[s,val] = v \wedge List(h[s,obs],h) = h[s,O]$$
$$Obs(o,s,v,h) \iff GO(s,h) \wedge GO(o,h) \wedge h[o,cache] = v \wedge h[o,sub] = s$$
$$SubObs(s,v,h) \iff Sub(s,v,h) \wedge (\forall o:\textbf{ref} \bullet GO(o,h) \wedge h[s,O][o] \implies Obs(o,s,v,h))$$

The specification of $update$, omitting effects $\textbf{wr}\ \langle this\rangle.val$ and $\textbf{wr}\ this.O.cache$, is

$$\textbf{requires } SubObs(this,val,Heap) \wedge GO(this,Heap)$$
$$\textbf{ensures } SubObs(this,n,Heap)$$

The implementation of $update$ uses local variables $o, r1$. As observers in the subject's (i.e., $this$'s) list of observers get notified they are put in region $r1$. This leads to the loop invariant that notified observers are up to date:

$$\forall p:\textbf{ref} \bullet GO(p,Heap) \wedge r1[p] \implies Obs(p,this,n,Heap)$$

Another loop invariant says region $this.O$ comprises $r1$ together with $List(o,Heap)$ which is the region containing objects yet to be notified.

$$List(o,Heap) \cup r1 = Heap[this,O]$$

The verification goes through, provided we include the equality axiom listed third in Fig. 3. It actually follows from the first two axioms using induction but here we work in pure first order logic (in which reachability is not finitely axiomatizable, cf. [9]). Note also the apparently superflous variable r in the second axiom. It is needed for triggering (Sect. 7).

We consider a client that creates two Subjects and updates one:

$$sub0 := \textbf{new } Subject();\ obs0 := \textbf{new } Observer(sub0);\ obs1 := \textbf{new } Observer(sub0);$$
$$sub := \textbf{new } Subject();\ obs := \textbf{new } Observer(sub);\ \textbf{call } sub0.update(5);$$

We are able to verify the following postcondition for the client:

$$DisjointSet(Heap[sub0,O], Heap[sub,O]) \wedge SubObs(sub0,5,Heap) \wedge SubObs(sub,0,Heap)$$

These assertions say that region $sub0.O$ is disjoint from $sub.O$ and updating the internal state of $sub0$ has no effect on the internal state of sub. The key links: condition $Obs(o,s,v,h)$ implies that $o.sub = s$ and thus $SubObs(s,v,h)$ implies that every o in $s.O$ has $o.sub = s$. This is much like an encoding of ownership (as pointed out in [4] and used in VCC [23]) and it implies that if $o.sub \neq o'.sub$ then $o.sub.O$ is disjoint from $o'.sub.O$.

BoogiePL is modular in the sense that in verifying this client code, the verifier uses only the specifications of the constructors and other methods. The next section considers modularity in more depth.

6 Abstraction and Hiding Examples

A flaw of the preceding Subject/Observer specification is that it mixes conditions on the internal data structures (the list) with those of interest to clients (including O which could be a model field). In Sect. 6.1 we factor apart the specification so that one part can be hidden from clients, in the manner of Hoare's treatment of invariants [16]. Hiding in this fashion introduces a potentially unsound mis-match between the specs used to verify invocations of a method and those with respect to which its implementation is verified. Such a mis-match can be justified by encapsulation, one technique for which is ownership: roughly, the invariant depends only on owned objects and clients are prevented from writing them.

A formal treatment of hiding, however, is beyond the scope of this paper. The interested reader is encouraged to consult [4] and [20] which address not only hiding but also encapsulation at the granularity of object clusters, as is needed for the Subject/Observer example. The use of regions for cluster encapsulation is similar to the use of regions for ownership.

In Sect. 6.2 we explore the use of a field to hold owned objects. The alternative to hiding is abstraction: Instead of a mis-match between the client specification of a method and its verification conditions, the idea is for client specifications to mention the invariant, but treated opaquely as a pure method —of which the owned objects are the footprint.

6.1 Hiding

As suggested in [4], suppose we put classes $Subject$ and $Observer$ together in a module, giving method $register$ module scope while the other methods are public. Sect. 5 considered an invariant, $SubObs$, that pertains to a single Subject and its Observers. Let us factor it into the externally visible part, $SubObsX$, and a hidden part, $SubObsH$, used only in verification of the implementations of the $Subject$ and $Observer$ methods.

$$SubX(s,v,h) \iff GO(s,h) \wedge h[s,val] = v$$
$$SubObsX(s,v,h) \iff SubX(s,v,h) \wedge \forall o : \mathbf{ref} \in h[s,O] \bullet Obs(o,s,v,h)$$

We verified client code, including the previous example, using $SubX$ and $SubObsX$ in place of Sub and $SubObs$ in the method specifications (excepting method $register$ which would be module scoped and not used by clients). The client verifications were

successful —confirming that they rely on the disjointness reasoning focused on the O field and not on properties of the list data structure.

The implementations of the methods of $Subject$ and $Observer$ are verified using specifications that use not only $SubObsX$ but also $SubObsH$ where

$$SubObsH(s,v,h) \iff SubH(s,h) \wedge SubObsX(s,v,h)$$
$$SubH(s,h) \iff List(h[s,obs],h) = h[s,O]$$

These verifications go through; they merely repackage the earlier specifications.

To cater for a second order frame rule to justify hiding, Banerjee et al [4] propose to hide a single "module invariant" that for this example could be the conjunction of $\forall s : \mathbf{ref} \bullet SubObsH(s,val)$ and

$$\forall p : \mathbf{ref}, s : \mathbf{ref} \bullet p \in Heap[s,O] \wedge Heap[p,sub] \neq \mathbf{null} \implies Heap[p,sub] = s$$

We added these as pre- and post-condition for each method, which entails disjointness reasoning with respect to arbitrary other Subjects. The verifications succeed.

6.2 Ownership and Abstraction

Although the clients considered above are well behaved, the proposed hiding is actually unsound since client code like $s.obs := s.obs.nxt$ could falsify $SubH(s)$. Some form of encapsulation is needed to preclude such clients. We now consider ownership, a popular device for achieving heap encapsulation. However, we do not delve into the justification of invariant hiding. Instead, we consider ownership for the alternative to hiding: abstraction. Abstraction has been used for invariants by Müller [19] (model fields), Parkinson [10] (existentially quantified predicates), and others. Smans *et al.* [24] use a pure boolean method $invariant()$ to abstract the invariant in specifications.

Like model fields, pure methods are also useful for properties other than invariants. We explore pure methods and ownership in an example of Smans et al [24]. A stack is implemented in terms of an $ArrayList$, a dynamically resizable array which implements its functionality with a primitive fixed-size array. The interface for $ArrayList$ allows elements to be added at the end of the array, to retrieve or delete the element stored at a specific index, and to query the current size of the array. The interface for $Stack$ is $push$, pop, $empty$. In addition, $Stack$ provides a pure method, $size$, that returns the current number of elements in the stack. This is used in the postconditions of the stack's constructor, which says that its value is zero, and the method $push$, which says $\#size() = \mathbf{old}(\#size()) + 1$.

Now $size$ is implemented using the internal representation of the stack. This internal representation consists of an owned object, the $ArrayList$, and its representation. Field *fp* will hold references to these, and method $size$ is specified to have read effect **rd** $this.fp.$any. The write effect of methods $push$ and pop is **wr** $this.fp.f$.

An interesting part of the example is the verification of the method $switch$ that exchanges the representations of two stacks. The specifications for most mutators would allow footprints to grow but only by fresh objects, but that is not the case for $switch$. Their collective footprint, however, need not change at all. We choose a specification

```
procedure switch(this:ref, other:ref)
  free requires GO(this, Heap) ∧ (other = null ∨ GO(other, Heap));
  requires other ≠ null ∧ Inv(Heap, this) ∧ Inv(Heap, other);
  requires DisjointSet(Heap[this,fp], Heap[other,fp]);
  ensures Inv(Heap, this) ∧ Inv(Heap, other);
  ensures DisjointSet(Heap[this,fp], Heap[other,fp]);
  ensures #size(Heap, this) = #size(old(Heap), other);
  ensures #size(Heap, other) = #size(old(Heap), this);
  ensures DifferenceIsNew(old(Heap), old(Heap)[this,fp] ∪ old(Heap)[other,fp],
                          Heap[this,fp] ∪ Heap[other,fp]);
  //write effect: wr fp•any, other.fp•any
  ensures ∀p:ref, f:field • GO(p, old(Heap)) ⟹
          old(Heap)[p,f] = Heap[p,f] ∨ old(Heap)[this,fp][p] ∨ old(Heap)[other,fp][p];
```

Fig. 4. Specification for Stack switch

```
procedure switch(this:ref, other:ref)
{ var tmp : ref;
  tmp := Heap[this, contents];
  Heap[this, contents] := Heap[other, contents];
  Heap[other, contents] := tmp;
  Heap[this,fp] := {this} ∪ {Heap[this, contents]}
                   ∪Heap[Heap[this, contents], ArrayList.footprint];
  Heap[other,fp] := {other} ∪ {Heap[other, contents]}
                   ∪Heap[Heap[other, contents], ArrayList.footprint];
  assume GH(Heap);
}
```

Fig. 5. Implementation for Stack switch

(Fig. 4) that allows the collective footprint to grow with fresh objects, to cater for benevolent side effects that the *ArrayList* might have. Note the free precondition guaranteed by the language semantics: the receiver of an instance method is non-null and allocated, while the parameter that would have been explicit in the source language is guaranteed to be allocated only if it is not null. However, we chose to add the explicit precondition that it be non-null. Requiring the two stacks to not be aliased is not enough: we need that their footprints are disjoint. We found the verifier is able to track this for newly allocated stacks.

The postcondition that encodes the write effect just says that all fields in the heap retain their old values unless the field is in an object in the footprint of either of the parameters.

We consider client reasoning using the following code:

```
var s1 : ref, s2 : ref; s1 := new Stack(); s2 := new Stack();
assert #size(Heap, s1) = 0 ∧ #size(Heap, s2) = 0;
call push(s2, 5);
assert #size(Heap, s1) = 0 ∧ #size(Heap, s2) = 1;
```

Consider the first assertion: the second conjunct, that #*size* is zero for $s2$ is directly from the constructor's postcondition (not shown). But for the first conjunct to hold it must be the case that the execution of the constructor for $s2$ did not affect the state that $s1$ depends on. This is done without revealing any of the details of the stack's implementation with the following axiom, which expresses that the effect of *size* is **rd** *this.fp*•any.

$$\begin{aligned}&\forall h, k: Hp, o: \mathbf{ref} \bullet \\ &\quad h[o,fp] = k[o,fp] \wedge (\forall p; \mathbf{ref}, f: \mathbf{field} \bullet h[o,fp][p] \implies h[p,f] = k[p,f]) \\ &\quad \implies \#size(h,o) = \#size(k,o);\end{aligned}$$

(Some *GO* conditions are elided.) Together with the default frame condition for constructors (Section 2), this is sufficient for the prover to be able to retain the knowledge about the size of the stack $s1$. The same holds for the second assert statement: the second conjunct is directly from the postcondition for $push$ while the first conjunct relies on knowing that calling $push$ on $s2$ cannot change the state that $s1$ depends on.

The above axiom is sound as long as *size* does not depend on the field of any object which is not contained in the stack's footprint. Utility of the axiom depends on how easy it is to determine that two states satisfy the antecedent, which in turn depends on the encapsulation of the representation objects.

7 Experiences with Prover

Verification failure might be due to (a) weak program specifications (pre- and postconditions), (b) weak loop invariant, and (c) incompleteness of triggers. In the first two cases one receives limited feedback from counterexamples.

A trigger of a universal quantifier is a set of expressions that determines how the SMT solver instantiates the quantifier. In theory the SMT solver can instantiate a universal quantifier with any ground term whatsoever. In practice it is better to limit the number of instantiations in some way so that the solver considers only a finite set of ground instantiations from its e-graph data structure (which is used to provide matching up to equivalence [12]). For example, consider the second axiom in Fig. 3 with the trigger restored:

$$\begin{aligned}&\forall o: \mathbf{ref}, h: Hp, r: [\mathbf{ref}]\mathbf{bool} \bullet \{List(o,h) = r\} \\ &\qquad o \neq \mathbf{null} \implies List(o,h) = \{o\} \cup List(h[o,nxt],h)\end{aligned}$$

This trigger instructs Z3 to instantiate the quantifier only with those o, h, r for which there is a term $List(o,h) = r$ in its e-graph.

Choosing a proper trigger is a craft. Had we removed the apparently redundant bound variable r and used $List(o,h)$ as a trigger, a matching loop would have arisen because the quantifier could be instantiated by any of $List(o,h)$, $List(o.nxt,h), \ldots$ We tried experimenting with the trigger $List(o.nxt,h)$ instead but this still did not suffice to verify the invariant $(List(o,h) \cup r1 = Heap[this, O])$ (see Sect. 5). Finally after much experimentation and with help from more experienced colleagues we arrived at the trigger $List(o,h) = r$: the intuition was that we needed one instantiation of the quantified variable r to be $Heap[this, O]$.

At the end of Sect. 1 we posed the question: Is a first order axiomatization of set theory effective for automating verification of programs and specifications that encode footprints in ghost state? We have to answer no, unless one is an expert at triggering quantifiers or has expert colleagues close by. The need for good triggers prevented us from introducing an abstract data type for regions in our specifications.

It is sobering to find out how easy it is to unintentionally introduce inconsistencies into the axiomatization. Boogie's `smoke` option creates multiple verification conditions, one per program path, then injects the statement **assert false** at the end of each path, converts all existing assert statements into assume statements and passes the result to the theorem prover. If the prover is able to establish the **false** assertion then the facts along that path reveal inconsistencies. Our experience has been that this is the single most valuable option in Boogie.

8 Related Work

The VCC project [23] is focused on the verification of low-level systems code written in C. It uses regions (sets) to structure the flat address space that the C memory model provides access to. Regions are disjoint sets of memory locations: pointers in C are modeled as offsets into a region. Footprints (read/write effects) are specified using (pure) functions, as opposed to our use of a designated field.

Smans et al [24] report on extensions of the Spec# tool to support dynamic frames where pure methods are used to represent footprints. In this work (and in [17]), footprints are sets of locations, where a *location* pairs an object reference with a field name. The language includes a notation, $\&e.f$, for the location (o,f) where o is the value of expression e; this is needed not only in specifications but in the ghost assignments that instrument code. This treatment offers more fine-grained expressiveness; in particular, it provides for abstraction over field names. Region logic needs a separate means to abstract over field names (notation "`any`" or model fields), but its instrumentation code can be written in Java and C# without need for the & operator (using classes like `Collection<Object>` for reference sets). It would be interesting to see whether the static analysis for framing in region logic [4] can be adapted to location sets. Besides checking write effects (using two-state postconditions as in our work), Smans *et al.* [24] check read effects in a small-step way: for each primitive expression that reads, a verification condition is generated that says the state read is within the specified read footprint. Our experiments did not check read effects. Modular reasoning is enforced in [24] like in our Sect. 6.1: In reasoning about code inside a module, the verifier is given an axiom that connects a pure method with its implementation. In reasoning about code outside a module, the verifier is given axioms about the frame of a pure method as well as its postcondition, but not its implementation.

Another difference between our work and [24] is our use of fields rather than pure methods to represent footprints. Fields are interpreted within the theorem prover so that reasoning is potentially more precise than for pure methods, though sometimes at the cost of more ghost assignments (e.g., Fig. 5). Some of the pure methods in [24] are recursively defined, which requires care [11]. Finally, they use conditional effects, for which we did not find a need.

Related work on ownership is discussed in [4].

9 Conclusion

We used BoogiePL to specify and verify several examples using effect specifications in the style of Region Logic [4], inspired by the dynamic frames of Kassios [17]. The standard theories of the underlying SMT solver, Z3, were augmented with some axioms for set theory. Bertrand Russell famously said the advantages of postulation are those of theft, but we found ample opportunity for honest toil in the instrumentation of axioms with triggers. Our toil would have been far greater without help from Rustan Leino, Peter Müller, and Michał Moskal. Nonetheless, we read our results as confirming the efficacy of effect specifications using ghost fields and auxiliaries holding just sets of references, lightweight machinery indeed.

The BoogiePL tool defines an axiomatic semantics for the BoogiePL programming language, insofar as it generates first order verification conditions, and we augmented that semantics with a few assumptions intended to express additional invariants of source language like Java or C#. We are unaware of any formal connection between BoogiePL and an actual language implementation. In his talk at VSTTE'05, J Moore argued that "we should build a [verification] system in some programming language for which we have a mechanically supported formal semantics and a mechanically supported reasoning engine – and [...] be able at least to state within the system what it means for our system to be correct. If you are working on a smaller piece of the problem – if you are building a system whose expressive power and implementation is beyond the scope of your own work – then you should find somebody to deal with the problem you are creating!". One use for a syntax-directed proof system like region logic is as bridge between verification conditions and the underlying semantics. The logic has been proved sound, on paper, with respect to a denotational semantics simplified from a Java/JML model that has been encoded by deep embedding in PVS [18]. Region logic has been extended with a second order frame rule which captures hiding of invariants and has been proved admissible, on paper [20]. Given the promising experiences by ourselves and others with dynamic framing, we see the remaining gaps as well worth bridging.

All example programs have been verified in Boogie version 0.90. Complete listings of these and a number of other verified programs are available online and in the accompanying technical report [3].

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL, Extended version available as KSU CIS-TR-2005-1 (2006)
2. Amtoft, T., Hatcliff, J., Rodriguez, E., Robby, H.J., Greve, D.: Specification and checking of software contracts for conditional information flow. In: Cuellar, J., Maibaum, T.S.E. (eds.) FM 2008. LNCS, vol. 5014. Springer, Heidelberg (2008)
3. Banerjee, A., Barnett, M., Naumann, D.A.: Boogie meets regions: a verification experience report (extended version). Technical Report MSR-TR-2008-79, Microsoft Research (2008)
4. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411 (2008)
5. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)

6. Barnett, M., De Line, R., Jacobs, B., Fähndrich, M., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# programming system: Challenges and directions. In: Verified Software: Theories, Tools, and Experiments (VSTTE) (2005)
7. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE (2005)
8. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)
9. Beckert, B., Trentelman, K.: Second-order principles in specification languages for object-oriented programs. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 154–168. Springer, Heidelberg (2005)
10. Bierman, G., Parkinson, M.: Separation logic and abstraction. In: POPL, pp. 247–258 (2005)
11. Darvas, Á., Müller, P.: Reasoning about method calls in interface specifications. Journal of Object Technology 5(5), 59–85 (2006)
12. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
13. De Line, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (March 2005)
14. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
15. Hehner, E.C.R.: Predicative programming part I. Commun. ACM 27, 134–143 (1984)
16. Hoare, C.A.R.: Proofs of correctness of data representations. Acta Inf 1, 271–281 (1972)
17. Kassios, I.T.: Dynamic framing: Support for framing, dependencies and sharing without restriction. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
18. Leavens, G.T., Naumann, D.A., Rosenberg, S.: Preliminary definition of core JML. Technical Report CS Report 2006-07, Stevens Institute of Technology (2006)
19. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
20. Naumann, D.A.: An admissible second order frame rule in region logic. Technical Report CS Report 2008-02, Stevens Institute of Technology (2008)
21. O'Hearn, P., Yang, H., Reynolds, J.: Separation and information hiding. In: POPL, pp. 268–280 (2004)
22. Parkinson, M.: Class invariants: the end of the road. In: International Workshop on Aliasing, Confinement and Ownership (2007)
23. Schulte, W.: Building a verifying compiler for C. Presentation at Dagstuhl Seminar 08061 for Types, Logics and Semantics of State (2008)
24. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for Java-like programs based on dynamic frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961. Springer, Heidelberg (2008)

Flexible Immutability with Frozen Objects

K. Rustan M. Leino[1], Peter Müller[1], and Angela Wallenburg[2]

[1] Microsoft Research
{leino,mueller}@microsoft.com
[2] Chalmers University of Technology and Göteborg University
angelaw@chalmers.se

Abstract. Object immutability is a familiar concept that allows safe sharing of objects. Existing language support for immutability is based on immutable classes. However, class-based approaches are restrictive because programmers can neither make instances of arbitrary classes immutable, nor can they control when an instance becomes immutable. These restrictions prevent many interesting applications where objects of mutable classes go through a number of modifications before they become immutable.

This paper presents a flexible technique to enforce the immutability of individual objects by transferring their ownership to a special freezer object, which prevents further modification. The paper demonstrates how immutability facilitates program verification by extending the Boogie methodology for object invariants to immutable objects. The technique is based on Spec#'s dynamic ownership, but the concepts also apply to other ownership systems that support transfer.

1 Introduction

Object immutability is a familiar concept that allows safe sharing of objects. For instance, immutable objects can be accessed concurrently without locking, and the absence of state changes can simplify reasoning about pointer structures.

A simple recipe for implementing immutable objects is the following: (1) Encapsulate the state of the immutable object by hiding all mutable fields from clients. (2) Do not provide any methods in the immutable object that modify the state of their receiver. (3) Encapsulate the mutable sub-objects of immutable aggregate objects by ensuring that references to mutable sub-objects do not escape from the aggregate.

This recipe is for instance applied in Java's *String* class: Its fields are private or final, methods that manipulate strings create new instances, but do not modify their receiver, and methods do not leak references to the (mutable) array that forms the internal representation of a string. The recipe is also amenable to static checking [13].

Even though this recipe is useful for value objects such as strings, it is too restrictive for many advanced applications of immutability. First, the recipe restricts *when an object becomes immutable*. Since immutable objects must not have mutating methods (rule 2), immutable objects must be fully initialized

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 192–208, 2008.

when their constructor terminates. This requirement prevents common initialization schemes such as multi-phase initialization. Second, the recipe restricts *the classes whose instances may be used as immutable objects*. Most classes in a program violate rule 2 of the recipe because they provide mutating methods.

```
using Map := Graph⟨City⟩ ;

class City {
  // some state, e.g. population
  City(String name)
  { /* code omitted */ }
}

class Graph⟨N⟩ {
  void AddNode(N n)
    modifies this.∗ ;
  { /* code omitted */ }
  void AddEdge(N from, N to)
    modifies this.∗ ;
  { /* code omitted */ }
  bool HasEdge(N from, N to)
  { /* code omitted */ }
  List⟨N⟩? Path(N from, N to)
  { /* code omitted */ }
}

class Metro {
  void AddTrainLines(Map map) ;
    modifies map.∗ ;
  { /* code omitted */ }
  void AddBusLines(Map map) ;
    modifies map.∗ ;
  { /* code omitted */ }
}
```

```
class Traveler {
  frozen Map map ;
  City current, next ;
  invariant map.HasEdge(current, next) ;
  Traveler(frozen Map map, City current)
  {
    this.map := map ;
    this.current := current ;
    this.next := current ;
  }
  void GoForward(City target)
    modifies this.∗ ;
  {
    List⟨City⟩? route ;
    route := map.Path(current, target) ;
    expose (this) {
      if (route = null) next := current ;
      else next := route.ItemAt(1) ;
    }
  }
}

class Main {
  void Setup(Metro metro)
  {
    Map map := new Map() ;
    metro.AddTrainLines(map) ;
    metro.AddBusLines(map) ;
    freeze map ;
    City c0 := new City("Oslo") ;
    City c1 := new City("Rome") ;
    Traveler t0 := new Traveler(map, c0) ;
    Traveler t1 := new Traveler(map, c1) ;
    /* ... */
  }
}
```

Fig. 1. Running example in a Spec#-like language. We assume that reference types are non-null types unless indicated otherwise by appending a question mark. The **using** directive introduces a synonym for a type. Frame specifications using **modifies** clauses indicate the potential side effects of a method. The **frozen** modifier as well as the **freeze** and **expose** statements are part of our methodology and will be explained later in the text.

Nevertheless, it is often useful to have immutable instances of such classes, for instance, immutable instances of general collections or, similarly, immutable arrays. The standard workaround for this limitation of the recipe is to wrap the supposedly-immutable instance or array in an immutable object that actually does follow the recipe. However, this workaround requires extra code to delegate calls to the wrapped object and also requires very careful design to ensure that there are no aliases to the wrapped object after it has been wrapped.

Example. We illustrate the limitations of the above recipe as well as our approach using the running example in Fig. 1. The example models travelers (class *Traveler*) who navigate using a map, which is a graph whose nodes are cities (abbreviated by type name *Map*). The map is shared by all travelers. We use an immutable map to allow *Traveler* to maintain a non-trivial invariant, namely that the map contains an edge between the traveler's current city and the next city on their route.

This example does not follow the above recipe. First, the map undergoes a complex initialization phase. After it is created in method *Setup* (class *Main*), it is passed as argument to two methods of class *Metro*, where more edges are added to the map. Only after these additions does the map become immutable. Such complex initialization phases occur frequently. For instance, nodes of an abstract syntax tree are mutated during resolution and type checking, but are often immutable afterwards. Second, the map is an instance of class *Graph*, which provides mutating methods such as *AddEdge*. Nevertheless, sharing of a map among travelers should be permitted, because travelers do not call these mutating methods.

Approach and Contributions. In this paper, we present a programming methodology that gives programmers the flexibility to decide when an object becomes immutable and also supports immutable instances of mutable classes.

Our work builds on the Boogie methodology for the verification of object invariants [2,14]. This methodology arranges objects in an ownership hierarchy and controls modifications of objects. It imposes the rule that an object must be *exposed* before its fields can be modified, and the rule that an object can be exposed only after its owner has been exposed. These two rules guarantee that modifications of objects are always initiated from the root of an ownership tree. Using this methodology, we can support immutable objects using three key ideas:

1. We use ownership to delimit the portion of an object's state that is immutable. In other words, the immutable state of an object o comprises the fields of o as well as the state of all objects (transitively) owned by o.
2. We provide a *freeze* operation that turns an object into an immutable object. This operation is an ownership transfer to a designated owner, called *freezer*, which cannot be referred to by the program. The freeze operation lets a programmer decide when an object becomes immutable.
3. We ensure that the freezer and all objects it owns, cannot be exposed. Consequently, the Boogie methodology prevents all modifications of the frozen objects, which makes them immutable even if they contain mutating methods or public fields.

In this paper, we present the details of this approach using Spec#'s dynamic ownership [14]. Our methodology also works with static ownership type systems that support ownership transfer [1,7,21].

Our work is related to various type systems for immutable objects [5,13,23,25], which also use ownership to delimit the object state and to control modifications. However, our methodology builds on verification instead of a type system to provide extra flexibility that we needed in several verification case studies. We also extend the Boogie methodology to make use of immutable objects. In summary, the two main contributions of our work are:

1. A verification methodology that supports immutability on a per-object basis. We show that immutable types are a special case of our methodology, where each instance is frozen at a particular program point. Our methodology also supports immutable instances of mutable types such as arrays.
2. An extension to the Boogie methodology that (a) guarantees that immutable objects satisfy their invariants in all states and (b) allows invariants to depend on immutable shared objects. We present the axioms and specification idioms for this extension.

Outline. The next two sections (2 and 3) explain the details of frozen objects and show how they subsume immutable types. Sec. 4 presents applications of frozen objects for the verification of object invariants. Sec. 5 discusses the application of frozen objects in Spec#. We discuss related work in Sec. 6 and conclude in Sec. 7.

2 Frozen Objects

In this section, we provide the background on the Boogie methodology that is needed in the rest of the paper. Based on this methodology, we explain how objects can be frozen and how frozen objects are encoded in the program logic. Since it is orthogonal to immutability, we defer a discussion of subtyping until Sec. 5.

2.1 Background on Boogie Methodology

The Boogie methodology [14] enables the sound verification of object invariants in the presence of callbacks by tracking whether an object is *valid*, that is, its invariant is known to hold, or whether it is *mutable*, that is, its invariant is allowed to be broken. Whether an object is valid or not is stored in a boolean field inv. The Boogie methodology maintains the following program invariant in all execution states:

PI0: $(\forall\, o \bullet\ o.inv \Rightarrow Inv(o))$

Here and throughout, the quantifier reaches over all allocated, non-null objects, and $Inv(o)$ denotes the object invariant of object o.

New objects start off as mutable because their invariants have not yet been established. The program invariant is maintained by two fundamental rules: First, we enforce through a proof obligation that only fields of mutable objects can be assigned to; therefore, the assignments trivially maintain PI0. Second, the invariant $Inv(o)$ is checked before an object o becomes valid, that is, when $o.inv$ is changed to true.

Objects become valid at the end of their constructor. Moreover, programs can change the *inv* field through a designated **expose** block statement. **expose** (o) $\{S\}$ sets $o.inv$ to false, thereby enabling modifications of o's fields. Then it executes statement S, checks o's invariant, and finally sets $o.inv$ back to true.

Aggregate objects are handled using ownership [8]. The invariant of an aggregate object o may depend on fields of another object x if x is owned by o[1]. The Boogie methodology enforces that the objects owned by a valid object o are also valid:

$$\text{PI1: } (\forall\, o, x \bullet\ x.owner = o \wedge o.inv \Rightarrow x.inv\ )$$

This program invariant is enforced by checking that the owner o of an object x is mutable when x is being exposed. It makes sure that program invariant $PI0$ is preserved even though modifications of x potentially break the invariant of o.

The Boogie methodology uses a dynamic encoding of ownership as opposed to a static type system. Each object has a field *owner* that holds a reference to the (single) owner object, or **null** if the object has no owner. The ownership relation is expressed by putting **rep** annotations on field declarations. For a field f, such an annotation gives rise to the implicit invariant $\mathbf{this}.f \neq \mathbf{null} \Rightarrow \mathbf{this}.f.owner = \mathbf{this}$. However, the *owner* field must not be used in explicit object invariants.

The *owner* field of an object x can be set to an object o by the special statement **transfer** x **to** o. Since the transfer might break the implicit ownership invariant of the previous owner, the **transfer** statement requires x's previous owner, if any, to be mutable. This requirement ensures that $PI0$ is maintained. To maintain $PI1$, **transfer** requires that x be valid or that the new owner o be **null** or mutable.

We define the semantics of the **expose** and **transfer** statements by translating them into the pseudo code shown in Fig. 2. Proving the correctness of a program amounts to statically verifying that the program does not abort due to a violated assertion. One can use an appropriate program logic to show that the assertions hold.

A typical method reads the state of its receiver and parameters—either by inspecting the state or by calling methods that do—and expects the invariants of these objects to hold. Therefore, we define a default precondition for all methods that requires their receiver and parameters to be valid. A common way to satisfy this default precondition is to use ownership—objects owned by a valid object are also known to be valid ($PI1$).

[1] The Boogie methodology supports several other kinds of multi-object invariants. We discuss invariants over peers [14] in Sec. 5. An extension of our methodology to history invariants [16] is straightforward.

If a method modifies the state of an object, it needs to declare that modification in the method's **modifies** clause. We then say that the method *mutates* the object and that the method is *mutating*. We say that a method with an empty **modifies** clause is *pure*. To perform the actual mutation of an object, a method needs to first expose the object, which requires the object's owner to be mutable. Therefore, for every object that a mutating method mutates (that is, that it lists in its **modifies** clause), we define an additional default precondition that requires that object's owner to be mutable.

In method *Main.Setup*, the call to the mutating method *AddTrainLines* satisfies its default precondition because *Setup*'s default precondition guarantees *metro* to be valid, and the **new** allocation constructs *map* to be valid and without an owner.

2.2 The Freeze Statement

In the Boogie methodology, an object can be modified only if the object and all its transitive owners are mutable. This offers a simple way of enforcing immutability. We introduce an object *freezer*, which is valid in all program states. The *freezer* object cannot be referred to in the program. In particular, it cannot be exposed. Consequently, objects owned by the *freezer*, called *frozen objects*, also cannot be exposed (by the second assertion of **expose**, see Fig. 2) and, thus, cannot be modified (by the assertion for field updates, see Fig. 2).

We provide a statement **freeze** x, which takes a valid object x and sets $x.owner$ to the *freezer*. Since the *freezer* is always valid, the pseudo code for **freeze** is a bit simpler than for **transfer**, see Fig. 2. Freezing an object x affects only one field, namely $x.owner$. The operation maintains program invariant $PI0$, because the only object invariants that can depend on $x.owner$ are the (implicit) object invariants of x's owner, and the precondition checks that any such owner is mutable. It also maintains $PI1$, because $x.inv$ is asserted before x is assigned a new owner.

```
x.f := e  ≡
  assert x ≠ null ∧ ¬x.inv ;
  x.f := e  ;

expose (o) { S }  ≡
  assert o ≠ null ∧ o.inv ;
  assert o.owner ≠ null ⇒ ¬o.owner.inv ;
  o.inv := false ;
  S ;
  assert (∀ x • x.owner = o ⇒ x.inv ) ;
  assert Inv(o) ;
  o.inv := true ;
```

```
transfer x to o  ≡
  assert x ≠ null ;
  assert x.owner ≠ null ⇒
         ¬x.owner.inv ;
  assert x.inv ∨ o = null ∨ ¬o.inv ;
  x.owner := o ;

freeze x  ≡
  assert x ≠ null ;
  assert x.owner ≠ null ⇒
         ¬x.owner.inv ;
  assert x.inv ;
  x.owner := freezer ;
```

Fig. 2. Pseudo code for field update, **expose**, **transfer**, and **freeze**. The new **freeze** statement is a variation of ownership transfer and will be explained in Sec. 2.2.

The **freeze** statement allows programmers to choose when an object becomes immutable. For simple value objects such as strings, this will be at the end of the constructor. But the **freeze** statement may also occur later, for instance, after a complex initialization phase as illustrated by method *Main.Setup*. The **freeze** *map* operation succeeds because *map* is valid and has no owner. Any subsequent attempt to expose *map* would fail because the second precondition of **expose** *map* does not hold: *map* does have an owner, namely the freezer, and the owner is valid. For the same reason, the object cannot be un-frozen by transferring it to another owner because the second assertion in the pseudo code for **transfer** would fail. In other words, the immutability of an object also applies to its *owner* field, and ownership transfer is just an update of *owner*. Consequently, *map* is permanently immutable once it has been frozen.

It is illustrative to discuss how frozen objects replace the recipe for immutable objects presented in the introduction. Rule 1 becomes dispensable because a frozen object cannot be exposed and, thus, its fields cannot be updated. For the same reason, the default precondition of mutating methods cannot be established, which makes rule 2 dispensable. This allows programs to create immutable instances of any class, even those that provide mutating methods such as *Graph*. Finally, rule 3 becomes dispensable because the immutability of a frozen object also applies transitively to the objects it owns. Assume for instance that a *Graph* is represented by a set of *Node* objects, which are owned by the *Graph* object. The nodes of a frozen *Graph* object are transitively owned by the freezer. Consequently, they cannot be exposed, which makes them immutable, even if they are leaked outside the graph structure.

2.3 Writing Specifications about Frozen Objects

To exploit properties of frozen objects during verification, specifications need to express which objects are frozen. For this purpose, we introduce a boolean function $\mathit{frozen}(o, h)$ that yields whether an object o is (transitively) owned by the freezer in heap h. We omit the heap parameter whenever it is clear from the context.

Instead of using *frozen* directly in specifications, we allow fields, method parameters, and method results to be declared with the modifier **frozen**. For a field f, this modifier gives rise to the implicit invariant $\mathbf{this}.f \neq \mathbf{null} \Rightarrow \mathit{frozen}(\mathbf{this}.f)$. This invariant, like all other object invariants, is checked at the end of constructors and **expose** blocks. For method parameters and results, we introduce analogous pre- and postconditions. For instance, class *Traveler* uses the **frozen** modifier in the declaration of field *map* and the constructor's first parameter.

For the soundness of the Boogie methodology, it is essential that each object invariant is admissible, that is, that it does not compromise program invariant $PI0$. Since the *owner* field of a frozen object x is immutable, $\mathit{frozen}(x)$ can never change from true to false. Thus, the implicit invariant for a frozen field f in an object o can be broken only by updating $o.f$. This update requires o to be mutable and, thus, preserves $PI0$.

2.4 Formal Encoding

The encoding of frozen objects in a program logic formalizes which objects are frozen and that frozen objects are immutable.

Frozen Objects. To capture the first aspect, one could define $frozen(o, h)$ to hold if and only if o is transitively owned by the freezer in heap h. However, automatic theorem provers such as Simplify and Z3 perform poorly on transitive closure. Therefore, we provide a weaker axiomatization of *frozen*, which is sufficient for practical examples: (a) The freezer itself is frozen. (b) Objects directly owned by a frozen object are also frozen:

$$(\forall\, h \bullet\ frozen(freezer, h)\ ) \tag{1}$$

$$(\forall\, o, h \bullet\ o.owner \neq \mathbf{null} \wedge frozen(h[o, owner], h) \Rightarrow frozen(o, h)\ ) \tag{2}$$

$h[o, owner]$ denotes the value held by field *owner* of object o in heap h. We abbreviate this notation by $o.owner$ when the heap is clear from the context.

Axioms (1) and (2), in combination with the pseudo code for the **freeze** statement and the **frozen** modifier explained in the previous subsection, allow one to prove that certain objects are frozen in a given heap. For instance, one can prove that $frozen(map)$ holds in the heap after the execution of **freeze** *map* in method *Setup*.

We have argued above that an object that is frozen in a heap h remains frozen in all successor heaps h' of h. We encode this property using a relation $HeapSucc(h', h)$ that expresses that heap h' is a successor of heap h. For every object allocation and field update, we add an assumption that the resulting heap is a successor of the initial heap. Moreover, we add a postcondition to each method and constructor stating that the poststate heap is a successor of the prestate heap. This postcondition may be assumed by callers, but need not be proven for method or constructor implementations. Using *HeapSucc*, we can now state that frozen objects remain frozen by the following axiom:

$$(\forall\, o, h, h' \bullet\ HeapSucc(h', h) \wedge frozen(o, h) \Rightarrow frozen(o, h')\ ) \tag{3}$$

In our example, this axiom allows us to prove that the allocation and initialization of *City* and *Traveler* objects in method *Setup* do not invalidate $frozen(map)$.

Immutability of Frozen Objects. We encode immutability of frozen objects by an axiom that says that the value of a field of a frozen object is a function of the object reference and the field name alone, and in particular does not depend on the heap:

$$(\forall\, o, f, h \bullet\ frozen(o, h) \Rightarrow h[o, f] = \#UltimateValue(o, f)\ ) \tag{4}$$

where $\#UltimateValue$ is an uninterpreted function symbol.

To illustrate the use of this axiom, we prove that the value of some field *map.nodes* is the same before and after allocating and initializing the *City* objects in method *Setup*. Let h and h' denote the heaps in these states. We

show this property by instantiating axiom (4) twice, once with map, $nodes$, and h, and once with map, $nodes$, and h'. As argued above, we can show $frozen(map, h)$ as well as $frozen(map, h')$. So we derive:

$$h[map, nodes] = \#UltimateValue(map, nodes) \quad \text{and}$$
$$h'[map, nodes] = \#UltimateValue(map, nodes)$$

which trivially implies the desired $h[map, nodes] = h'[map, nodes]$.

3 Immutable Types

With frozen objects, we can designate individual objects as being immutable from a particular time onward. Therefore, frozen objects are strictly more general than immutable types, where the immutability of an object is determined by its type and always occurs from the end of the constructor on. In this section, we show how frozen objects can be used to encode immutable types and explain the benefits of such an encoding.

3.1 Encoding Immutable Types

Let's assume that immutable classes are marked with a modifier **immutable**. We define the meaning of an **immutable** class C in terms of what we introduced in the previous section. An allocation $c := \textbf{new}\ C(\ldots)$ is immediately followed by an implicit statement **freeze** c. Therefore, every instance of class C is immutable once its constructor has terminated.

To make use of the immutability, for instance, for verification, we augment specifications as follows: Every parameter, receiver, and return value p of static type C, except the receiver of constructors (and other *delayed* arguments, if applicable [12]), gives rise to an implicit precondition (or postcondition, in the case of results) $p \neq \textbf{null} \Rightarrow frozen(p)$, as if p had been declared with the **frozen** modifier. Similarly, every field f of static type C gives rise to the implicit object invariant $f \neq \textbf{null} \Rightarrow frozen(f)$, as if the field had been declared with the **frozen** modifier. Finally, every local variable x of static type C gives rise to an implicit loop invariant $x \neq \textbf{null} \Rightarrow frozen(x)$ on every loop that modifies x. These implicit pre- and postconditions and invariants are checked in the same way that explicit ones are. These checks, for instance, prevent a program from passing an instance of C as a (**frozen**) parameter before the instance has been fully initialized and frozen.

We also allow the **immutable** modifier to be applied to interface types, with the same meaning as just described for **immutable** classes. A class or interface that extends or implements an **immutable** class or interface must itself be declared **immutable**; therefore the object stored in a variable of an **immutable** type is actually an instance of an **immutable** class.

3.2 Benefits

The current implementation of immutable types in Spec# is not yet based on frozen objects and, although our motivation had been different from theirs,

shares most virtues of the immutable-type design by Haack *et al.* [13]. In this subsection, we explain that our new design is not only more flexible, but has two additional major virtues compared to alternative designs.

A first virtue is that, rather than introducing the pre- and postconditions and invariants, as described in the previous subsection, it is tempting simply to encode an immutable type C by the following *Tantalizing Axiom*:

$$(\forall\, o, h \bullet\ o \in C \Rightarrow \mathit{frozen}(o, h)\ )$$

However, one has to be careful about the reach of o in this quantification, because an object of type C is allowed to undergo mutations (and in particular initialization) until the end of its construction. Various ways exist to exclude from the reach of the Tantalizing Axiom all objects currently being constructed. For example, one can design the programming language in such a way that the object being constructed does not come into being until values for all its fields have been computed (see, *e.g.*, Theta [18]). In languages like Spec# and Java, a constructor can assign to a field multiple times and can access the object while it is being constructed. To use the Tantalizing Axiom for such languages, one needs a stronger antecedent as well as some escape analysis in constructors. Defining this antecedent turned out to be complicated; we found that it was difficult to prevent the Tantalizing Axiom from kicking in too soon and interacting with other axioms in undesired ways. In contrast, the encoding of **immutable** types on top of frozen objects systematically introduces checks that objects of immutable types have reached their frozen state.

Another virtue of our encoding pertains to the knowledge that certain objects are not immutable. In a system where immutability is found only at the granularity of types, a checker must decide based on the static type of a target object o whether or not to allow an assignment to a field $o.f$, which in general depends on the dynamic type of o. For example, suppose the static type B of o were a mutable class and the program contained an immutable subclass C of B; then, a type-based analysis cannot determine precisely whether $o.f$ is allowed to be modified because at runtime, o might reference an (immutable) C object. A sensible solution is to impose a *One-Down Rule* that says that each immediate subclass T of *Object*—the root of the class hierarchy—must either decide to make T and all its subclasses mutable or to make them all immutable [13]. In our new encoding of immutable types, checking if an assignment to $o.f$ is legal depends on the dynamic state of object o, and in particular on the validity of o. Therefore, we do not need such a One-Down Rule and instead allow mutable classes to have immutable subclasses (but not vice versa).

4 Frozen Objects and Object Invariants

Frozen objects have many benefits for writing and reasoning about code. For instance, they simplify the development of correct multi-threaded code and the verification of contracts containing pure method calls. In this section, we illustrate how frozen objects extend the benefits of owned objects—guaranteed validity and support for multi-object invariants—to shared objects.

4.1 Proving Validity

As explained in Sec. 2.1, the Boogie methodology makes explicit whether an object invariant may be assumed to hold. While making this information explicit enables the sound verification of object invariants in the presence of callbacks, it also complicates verification. Methods have default preconditions that require the validity of their arguments, and callers have to live up to these preconditions.

For instance, the call to *map.Path* in method *Traveler.GoForward* leads to a proof obligation that *map* is valid. In the Boogie methodology, validity of an object typically follows from ownership. If the *Traveler* instance **this** owned **this**.*map* (that is, if *map* were a **rep** field), then the validity of **this**.*map* would follow from the validity of **this** by program invariant $PI1$. However, arranging for *Traveler* objects to own their maps prevents them from sharing one map, which requires cloning of *Map* objects. For immutable objects, this cloning is unnecessary and unnatural because sharing is actually safe.

We solve this problem by encoding in our program logic that a frozen object is always valid. That is, we add the following axiom:

$$(\forall\, o, h \bullet \mathit{frozen}(o, h) \;\Rightarrow\; h[o, \mathit{inv}]\;) \tag{5}$$

This axiom is justified because the **freeze** o statement requires o to be valid, and o cannot be exposed afterwards. Thus, o forever remains valid.

Axiom (5) allows one to show the validity of a frozen object without restricting sharing. For instance, from the implicit invariant for the **frozen** field *map*, we conclude that **this**.*map* is frozen in the prestate of method *GoForward*, and by axiom (5), we get *map.inv*. That is, we can share the *Map* instance among *Traveler* objects and nevertheless live up to the default preconditions of methods operating on the map.

4.2 Invariants over Frozen Objects

Object invariants in the Boogie methodology are constrained by *admissibility requirements*. For the basic methodology (ownership-based invariants [2,14]), an admissible invariant of an object o may depend on the state of o and of the objects (transitively) owned by o. This requirement ensures that all objects whose invariant is potentially broken by an update of $x.f$ are mutable such that program invariant $PI0$ is maintained.

While these ownership-based invariants enable modular verification of aggregate objects, they do not support sharing of objects. In our example, the invariant of class *Traveler* depends on the state of a *Traveler*'s *Map* instance and would, therefore, be admissible only if *map* was a **rep** field. As we have explained in the previous subsection, this would prevent *Traveler* objects from sharing a *Map* instance.

There are extensions to the basic Boogie methodology that support invariants over shared objects. However, these extensions restrict the classes of shared objects (visibility-based invariants [14]), restrict the invariants that one can write about shared objects [16], or complicate verification [3].

All variations of the Boogie methodology have in common that they restrict programs or invariants such that one can determine modularly all objects whose object invariant is potentially broken by an update of $x.f$; this allows one to impose proof obligations on the update that maintain program invariant $PI0$. Since frozen objects are immutable, their fields cannot be updated and, thus, maintaining object invariants over frozen objects is trivial.

To support such invariants, we relax the definition of admissible invariants of the basic Boogie methodology. In addition to the state of o and of the objects (transitively) owned by o, the invariant of an object o may now also depend on the state of frozen objects. This admissibility requirement can be checked syntactically by enforcing that only fields with the **rep** or **frozen** modifier can be dereferenced in an object invariant; such a syntactic check requires pure methods that are used in invariants to have read effect specifications [15].

In our example, the invariant of *Traveler* is admissible under the assumption that the method *HasEdge* reads only the state of its receiver and the objects owned by the receiver. Since field *map* is declared with the **frozen** modifier, we can conclude that the invariant depends only on the state of **this** and of frozen objects, namely the map referenced from **this**.*map*.

The discussion above illustrates why it is not easily possible to permit frozen objects to be un-frozen. Un-freezing an object x would require one to find all objects whose invariants depend on the state of x because these invariants might be broken by modifications of x. Finding these objects in a modular way is at best complicated [3].

We have not yet implemented frozen objects in Spec#, but we manually changed the Spec#'s encoding of the program in Fig. 1 to resemble the encoding in this paper. The Boogie tool verified the resulting program successfully.

5 Application to Spec#

The form of the Boogie methodology implemented in Spec# is more advanced than the basic form we have presented here. In this section, we discuss how we have extended our methodology to handle the three differences most relevant to this paper: updates without expose, subclasses, and peers.

Updates without Expose. One difference between the methodology we have presented here and what is implemented in Spec# is that Spec# is more lenient about when expose statements are needed. Instead of insisting that an object be mutable when one of its fields is updated, Spec# still allows the update if it maintains the object invariant. However, since object invariants of the owner may depend on the field being updated, and such object invariants might not be known in the current verification scope, Spec# will nevertheless check that the owner of the target object is mutable. Formally, for a field update of $o.f$, instead of the precondition $\neg o.inv$, Spec# actually checks:

$$\neg o.inv \vee ((o.owner \neq \textbf{null} \Rightarrow \neg o.owner.inv) \wedge Inv'(o))$$

where $Inv'(o)$ denotes the invariant of o in the heap after the update of $o.f$.

Since the freezer is always valid, this more lenient field-update precondition always fails for a frozen object o. Consequently, fields of frozen objects do not change.

Subclasses. A second difference is that we have ignored subtyping so far whereas Spec# handles it. Instead of using just one *inv* field per object and letting the expose statement take an object argument, Spec# essentially uses one *inv* field per object-type pair and refines the expose statement to operate on object-type pairs as well [17]. Moreover, an owner in Spec# is not just an object but an object-type pair [14]. The effect of this support on the present paper is that the freezer becomes a pair $(\mathit{freezer}, \mathit{Freezer})$, where *Freezer* denotes some fixed but arbitrary type. Also, the condition $x.\mathit{inv}$ appearing in the preconditions of **transfer** and **freeze** (Fig. 2) is replaced by:

$$(\forall\, T \bullet\ (x, T).\mathit{inv}\ )$$

Peers. The third difference is that Spec# many times considers not just single objects but groups of peer objects. Two objects are *peers* if they have the same owner [14]. More precisely, in Spec#, the *owner* field partitions objects into *peer groups*; the value of the *owner* field is either the object-type owner (ow, T) that owns the objects in the peer group or, if the objects in the peer group have no owner, takes on the value $(r, \bot)$ where r is a representative element (object) of the peer group. Having an *owner* value of the second kind corresponds to the condition that we have written $\mathit{owner} = \mathbf{null}$ elsewhere in this paper. Peer groups are similar to *clusters* [21] and allow the notion of peers even when objects are unowned.

In the presence of peer groups, a newly allocated object o starts off being unowned and belonging to a new peer group: $o.\mathit{owner} = (o, \bot)$. We redefine the statements **transfer** x **to** o and **freeze** x to transfer not just x but the entire peer group of x; so the assignment to $x.\mathit{owner}$ in the pseudo code for these statements (Fig. 2) is replaced by the same assignment for all peers of x. If the new owner in the **transfer** statement is specified as **null**, the right-hand side of the assignment is $(x, \bot)$. Also, the condition $x.\mathit{inv}$ appearing in the preconditions of these statements is replaced by:

$$(\forall\, p \bullet\ p.\mathit{owner} = x.\mathit{owner} \Rightarrow (\forall\, T \bullet\ (p, T).\mathit{inv}\ ))$$

which says that x is *peer valid*, meaning that x and all its peers are valid. (Note, though, that the precondition $o.\mathit{inv}$ of the **expose** statement remains what it is in Fig. 2, because the **expose** statement operates on a single object-type pair.)

In a similar way, Spec# does not use validity as the default method pre- and postconditions we described it in Sec. 2.1, but uses peer validity. This makes no difference for frozen objects, since frozen objects are always peer valid.

6 Related Work

Immutability is a fundamental concept with many applications. In this section, we discuss work related to the two main contributions of this paper, how to

enforce object immutability in imperative object-oriented languages and how to use immutable objects for verification.

Like our methodology, several type systems for immutability use ownership to delimit the state of an object. Boyapati [5] as well as Östlund *et al.* [23] present type systems that support immutable instances of mutable types and that let programmers decide when an object becomes immutable. Objects that will become immutable can only be referenced by unique references to make the transition to immutability type-safe. Since our methodology builds on verification rather than a type system, we do not need this restriction.

IGJ [25] uses Java's generic types to check immutability of classes, objects, and references. IGJ requires immutable objects to be fully initialized at the end of the constructor, which prevents complex initialization schemes. Oval [19] achieves per-object immutability by setting the owner to bottom. This is done when the object is created and, thus, not as flexible as frozen objects.

Haack *et al.* [13] present a type system for immutable types. As we have explained in detail in Sec. 3, our methodology is more flexible and, in fact, subsumes immutable types. A major virtue of Haack *et al.*'s type system is that it guarantees immutability even if some portions of the code (such as libraries) are not checked with the system. In contrast, we require all code to follow our methodology.

We presented our methodology in terms of Spec#'s dynamic ownership [14], but it can also be used with ownership type systems that support transfer such as Universes [21], External Uniqueness [7], or AliasJava [1] (even though AliasJava requires additional restrictions to prevent the modification of frozen objects through lent references).

Reference immutability like in Javari [24] or Universes [20] guarantees that certain read-only references are not used to modify an object. They allow safe sharing of objects. However, an object referenced through a read-only reference is still mutable and may be modified through other references. Thus, reference immutability does not have the same benefits for verification as immutable objects: objects referenced read-only cannot be assumed to be always valid, and object invariants must not depend on these objects [20]. In some of the examples we have encountered, we found use for an idiom similar to read-only references. We declared method parameters and fields as "peer or frozen". Like read-only references, such references cannot be used to modify the referenced object (because it might be frozen). However, unlike read-only references in Universes, the object can still be assumed to be valid (the validity of peers follows from a method's peer-validity default precondition).

Boyland has used fractional permissions to differentiate read capabilities from write capabilities [6]. By squandering a part of an object's permissions, by giving a fractional permission to the freezer, or by weakening a full permission to an existentially quantified permission (cf. [4]), one can effectively freeze an object.

Visible state invariants may be assumed to hold in the pre- and poststates of all method executions; callers do not have to show the validity of method parameters explicitly. Thus, verification techniques for visible state invariants [11,20] do not benefit from the fact that frozen objects are always valid. However, like

the Boogie methodology, these techniques can be extended to support invariants over shared, frozen objects.

Our notion of immutability forbids all mutations of a frozen object, even benevolent mutations that are not observable by clients, such as lazy initialization. Naumann's work on observational purity [22] shows how to check that mutations are benevolent; it can be combined with our methodology to make it applicable to more programs.

7 Conclusions

The concept of immutable objects is useful and widely used in programming. Advanced support of immutability in a programming language or system requires a fine level of granularity, in order to accommodate the variety of ways that a program selects which objects are to be immutable and just when each object becomes immutable. In this paper, we have presented *frozen objects* as a technique for specifying and verifying programs that use such immutability properties. Our technique guarantees that the fields of frozen objects do not change. It is based on object ownership, where a special freezer object is used as the (transitive) owner of all frozen objects. Though we have presented our solution in the context of dynamic ownership, frozen objects also work with any ownership type system with ownership transfer.

Frozen objects subsume immutable types; encoding immutable types on top of frozen objects leads to a better axiomatization in a verification logic than a direct encoding and is also less restrictive.

Our technique guarantees that frozen objects are *valid*, meaning that their object invariants hold. In fact, this ever-validity had been our original motivation for this work. Because their fields do not change, frozen objects can be shared in a carefree way, and we allow any object invariant to depend on the fields of frozen objects.

There are other intriguing applications of immutability. One is in support of pure methods (*e.g.*, [10,9,15]), which for frozen objects return values that are insensitive to the heap. Another is in conjunction with concurrency (*e.g.*, [13]), which can benefit from the carefree sharing that frozen objects offer.

In future work, we intend to implement frozen objects in Spec#. We would also like to explore how to relax immutability to permit certain modifications of otherwise frozen objects, for instance, by using monotonicity.

References

1. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, pp. 311–330. ACM Press, New York (2002)
2. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. JOT 3(6), 27–56 (2004), www.jot.fm

3. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
4. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, vol. 40(1), pp. 259–270. ACM, New York (2005)
5. Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. Ph.D., MIT (2004)
6. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
7. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 176–200. Springer, Heidelberg (2003)
8. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA. ACM SIGPLAN Notices, vol. 33(10) (1998)
9. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 336–351. Springer, Heidelberg (2007)
10. Darvas, Á., Müller, P.: Reasoning about method calls in interface specifications. JOT 5(5), 59–85 (2006)
11. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: A unified framework for verification techniques for object invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
12. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA. SIGPLAN Notices, vol. 42(10), pp. 337–350. ACM, New York (2007)
13. Haack, C., Poll, E., Schäfer, J., Schubert, A.: Immutable objects for a Java-like language. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 347–362. Springer, Heidelberg (2007)
14. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–516. Springer, Heidelberg (2004)
15. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008)
16. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 80–94. Springer, Heidelberg (2007)
17. Leino, K.R.M., Wallenburg, A.: Class-local object invariants. In: First India Software Engineering Conference (ISEC). ACM, New York (2008)
18. Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P., Myers, A.C.: Theta reference manual, preliminary version. Memo 88, Programming Methodology Group, MIT Laboratory for Computer Science (1995), http://www.pmg.lcs.mit.edu/Theta.html
19. Lu, Y., Potter, J., Xue, J.: Validity invariants and effects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 202–226. Springer, Heidelberg (2007)
20. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming 62, 253–286 (2006)
21. Müller, P., Rudich, A.: Ownership transfer in Universe Types. In: OOPSLA. SIGPLAN Notices, vol. 42(10), pp. 461–478. ACM, New York (2007)
22. Naumann, D.A.: Observational purity and encapsulation. TCS 376(3), 205–224 (2007)

23. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, uniqueness and immutability. In: IWACO (2007)
24. Tschantz, M.S., Ernst, M.D.: Javari: adding reference immutability to Java. In: OOPSLA. SIGPLAN Notices, vol. 40(10), pp. 211–230. ACM, New York (2005)
25. Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A., Ernst, M.D.: Object and reference immutability using java generics. In: ESEC-FSE, pp. 75–84. ACM, New York (2007)

The Verisoft Approach to Systems Verification

Eyad Alkassar[1,⋆], Mark A. Hillebrand[2,⋆⋆], Dirk Leinenbach[2,⋆,⋆⋆], Norbert W. Schirmer[2,⋆⋆], and Artem Starostin[1,⋆⋆⋆]

[1] Universität des Saarlandes
P.O. Box 15 11 50, 66041 Saarbrücken, Germany
{eyad,starostin}@wjpserver.cs.uni-sb.de
[2] German Research Center for Artificial Intelligence (DFKI)
P.O. Box 15 11 50, 66041 Saarbrücken, Germany
{mah,dirk.leinenbach,norbert.schirmer}@dfki.de

Abstract. The Verisoft project aims at the pervasive formal verification from the application layer over the system level software, comprising a microkernel and a compiler, down to the hardware. The different layers of the system give rise to various abstraction levels to conduct the reasoning steps efficiently. The lower the abstraction level the more details and invariants are necessary to ensure overall system correctness. Illustrated by a page-fault handler we discuss the layers and the trade-off between efficiency of reasoning at a more abstract layer versus the development of meta-theory to transfer the verification results between the layers.

1 Motivation and Challenges

The layer of system software confines the essential components of modern computer architectures. Any flaw up to this level has a decisive impact on the robustness, safety, and security of applications running on top of it. An operating system kernel that might fail to guarantee isolation of processes can hardly serve as a trustworthy computing basis to process security critical data. Hence, it is both a worthwhile and promising effort to design and verify the crucial system level parts by the most rigorous means.

Examining the system design up to the level of a microkernel, we typically have to deal with at least the following layers: hardware, assembler, and the C programming language. The different layers come along with a rise in abstraction regarding formal models and reasoning about them. However, the final goal is to provide objective evidence that the actual running system behaves correctly. The lower the layer the 'correctness theorem' holds on the better. Ideally, this is a theorem in the domain of physics. For computer science a transistor

⋆ Work was supported by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems'.
⋆⋆ Work was supported by the German Federal Ministry of Education and Research (BMBF) within the Verisoft project under grant 01 IS C38.
⋆⋆⋆ Work was supported by the International Max Planck Research School for Computer Science (IMPRS-CS).

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 209–224, 2008.

or gate-level hardware model is a realistic target to state the final correctness result. Employing higher abstraction levels to improve effectiveness of reasoning demands that we close the 'semantic gap' and bring the results down to the hardware level. This is the very idea of *pervasive* or *systems* verification [1,2]. In Verisoft every abstraction layer is justified by meta-theorems that allow transferring the results to the low-level models. All the development is mechanized in the uniform logical framework of the interactive theorem prover Isabelle/HOL and hence it is rigorously checked that all the results fit together.[1] The goal of this paper is to provide an informal overview of the different layers and their connection. This bird's eye view easily gets lost in detailed technical papers on parts of this work that were already or are simultaneously published.

Related Work. First attempts to use theorem provers to specify and even prove correct operating systems were made in the 1970ies in PSOS [3] and UCLA Secure Unix [4]. However, a missing or underdeveloped tool environment made mechanized verification futile. With the CLI stack [1], a new pioneering approach for pervasive systems verification was undertaken. The goal of this project was to build a system from verified, hierarchically stacked components. In extension to their seminal work the Verisoft project aims at a more realistic system architecture regarding both hardware and system software. In particular, devices are integrated into the Verisoft system stack. For realistic systems, this is already required for booting or scheduling in a microkernel. It is theoretically challenging, since devices are a concurrent source of computation and break the abstraction of sequential programs.

The project L4.verified [5] focuses on the verification of an efficient microkernel, rather than on formal pervasiveness, as no compiler correctness or an accurate device interaction is considered. The microkernel is implemented in a relatively large subset of C, including pointer arithmetic and an explicit low-level memory model [6]. So far, only portions of kernel code were reported to be verified and the virtual memory subsystem uses no demand paging [7]. Code verification relies on Verisoft's Hoare environment [8]. In the FLINT project, an assembly code verification framework is developed and code for context switching on a x86 architecture was formally proven [9]. A program logic for assembler code is presented, but no integration of results into high-level programming languages is undertaken. The VFiasco project [10] aims at the verification of the microkernel Fiasco implemented in a subset of C++ and embedded into PVS. There is no attempt to map the results to the machine level.

Overview. In Sect. 2 we give an overview of the Verisoft system stack and our approach towards pervasive verification. We proceed in Sect. 3 by introducing a language stack from a Hoare logic down to a low-level small-step semantics for C0, the C-like programming language used in Verisoft. The compiler verification detailed in Sect. 4 is the bridge between C0 and the machine model. In Sect. 5 we explain how machine-level entities can be made accessible from within the high-level Hoare logic based reasoning and Sect. 6 explains the integration of devices.

[1] Theory files are available at `http://www.verisoft.de/VerisoftRepository.html`.

In Sect. 7 we illustrate our approach with a page-fault handler implementing demand paging for memory virtualization. We conclude in Sect. 8.

2 Pervasiveness

In short, pervasive verification means that finally we obtain a correctness theorem at the lowest level of abstraction, the machine level. To avoid conducting all the verification at the machine level in the first place we introduce layers (like a C0 semantics and a Hoare logic) to improve the level of abstraction along with the performance of verification. Nevertheless, meta-theoretic theorems allow us to bring the verification results all the way down to the machine level, where they can be composed for an overall system correctness proof. Figure 1 depicts our system stack. The bottom layer is given by the gate-level description of the VAMP hardware, our hardware platform. A processor correctness result [11] links this to the layer of the instruction set architecture (ISA), where instructions and values are encoded as bit-vectors. The VAMP assembler layer formalizes the programmer's view on the machine and is a slight abstraction of the ISA layer, where the bit-vector encodings of the instructions and values are switched to an abstract data type and natural numbers, respectively. A straightforward simulation theorem connects these layers. Assembler computations are described by a small-step semantics. The high-level programming language C0 is also formalized with a small-step semantics and a compiler correctness result relates C0 computations to assembler computations. This is the junction where inline assembler code and concurrent device interaction can be merged into purely sequential C0 code. At the upper levels the inline assembler code

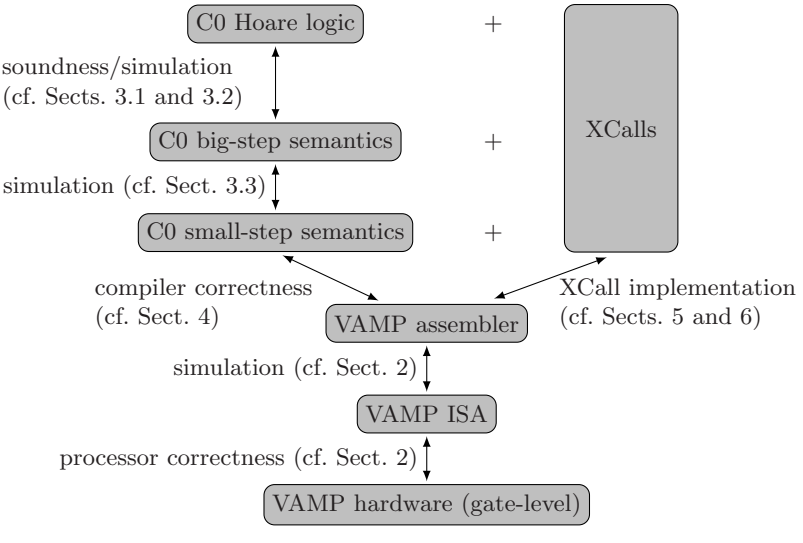


Fig. 1. System Stack

is abstracted to so-called XCalls. A big-step semantics serves as intermediate layer towards a Hoare logic, which is our main vehicle for reasoning about C0 programs. Soundness of the Hoare logic and simulation between the operational semantics allow us to transfer the Hoare triples down to the small-step semantics.

3 C0 Semantic Stack

The C0 programming language is a subset of C, designed to be expressive enough to allow the implementation of large parts of the system software, while remaining 'neat' to verify by restricting ourselves to a type-safe fragment without pointer arithmetic. In the lucky position not having to verify already existing C system code we could develop the code base ourselves in parallel with the specification and verification. Hence, it was reasonable to restrict ourselves to a subset of C to make the code verification less tedious and detailed, and thus make it faster. However, it is intrinsic to system level code to perform low-level architecture dependent operations. We clearly cannot avoid those parts without sacrificing any correspondence to real systems. Our approach is to encapsulate those tricky low-level parts in inline assembler code. Fortunately, these parts only make up a small fraction of the code base.

The C0 semantics plays a central role. On the one hand the Hoare logic has to be sound with respect to the C0 semantics, on the other hand it is a cornerstone of the compiler correctness theorem that links the formal C0 semantics to the machine semantics of the generated code. All the Hoare logic based code verification, as well as the C0 semantics, the compiler and the meta-theorems are mechanized in the theorem prover Isabelle/HOL. It is challenging to come up with a formalization that is equally well equipped to support both the verification of individual programs besides the meta-theory. For example, to specify and verify a C0 compiler it is natural to explicitly represent C0 types, such that the compiler can exploit this information. To reason about an individual C0 program however, this type information is not explicitly necessary or can be even obstructive. In C0 there is neither a means nor a need to access type information from within the program. Quite the opposite, the C0 programs we consider are all well-typed and C0 is a type safe language. Hence it is tempting to find a way to abstract from potential typing issues while reasoning about individual programs. In a typed logical framework like HOL this can be achieved by mapping programming language types for C0 variables directly to HOL types. The built in type inference of Isabelle/HOL then automatically takes care of typing. While this is a reasonable approach for a Hoare logic, it is not well-suited as a formalization of the C0 semantics itself and the accompanying meta-theory. In this context an explicit representation of programming language typing is necessary in order to even express things like 'type-safety'. These conflicting requirements on 'the' C0 semantics (together with further ones described below) is the reason we defined a complete C0 semantics stack. To preserve our goal of pervasive verification the semantics are linked together via meta-theorems.

A more fundamental difference in the semantic formalization is the decision between big-step and small-step semantics. C0 is a sequential language and even the target machine is a uni-processor architecture. Sequential reasoning and a big-step semantics seem to be adequate. However, the processor is not the only relevant hardware unit at the level of system software: devices run concurrently with the processor, and communication involves interrupt handling and memory-mapped I/O. Moreover, there may also be concurrency at the level of communicating user processes. Therefore we use a small-step semantics to handle concurrent aspects and a big-step semantics for the sequential parts.

The big-step semantics serves as a target for Hoare logic based reasoning: a Hoare triple is a 'big-step' program property relating the initial and final state. In general, a small-step semantics is more expressive than the big-step counterpart. It describes the complete trace of the computation rather then just the initial and final state. Moreover, it can be used to reason about non-terminating computations, whereas the big-step semantics does not distinguish between stuck computations and non-termination. A big-step semantics however is more abstract (e.g., an explicit frame-stack for procedures is unnecessary) and supplies a powerful proof principle: rule induction (i.e., induction on the depth of the derivation tree). The similar structure of both a big-step semantics and a Hoare calculus makes its soundness proof quite straightforward.

An orthogonal issue is the representation of programming language values. C0 offers aggregate values as arrays, structures, or combinations of both. For reasoning about C0 programs it is both sufficient and comfortable to keep this abstraction of aggregate values: a single 'memory cell' can contain an aggregate value of arbitrary size. The real memory of the target architecture however is byte addressable and aggregate values are stored in a consecutive sequence of bytes. We flatten aggregate values to byte sequences in the small-step semantics whereas we treat them as compact atomic entities in the big-step semantics.

Another aspect of aggregate values is their representation on the heap, in particular considering dynamic pointer structures. Aliasing is one of the key obstacles when reasoning about pointer programs. Any reduction of potential aliasing means a significant benefit for verification. Type-safety of C0 prohibits aliasing between pointers to different types. We exploit this invariant by switching to a different memory model in the Hoare logic. Instead of a single monolithic heap that stores every kind of value, we introduce a separate heap [12] for each field of a structure. Without further reasoning the model already rules out aliasing between different types or even different structure fields.

3.1 Simpl Hoare Logic

The Hoare logic environment [8], built on top of the Isabelle/HOL, was motivated by C0 but is by no means restricted to C0. It is a self-contained theory development for a quite generic model of a sequential imperative programming language called *Simpl*. A big-step semantics as well as a Hoare logic for both partial and total correctness is defined. Soundness and completeness is proven.

Theorem 1 (Soundness). *Every triple derived in the Hoare logic is valid with respect to the operational semantics.*

Theorem 2 (Completeness). *Every triple that is valid with respect to the operational semantics can be derived in the Hoare logic.*

Soundness is crucial for pervasive verification in order to formally link the results from the Hoare logic to the operational semantics. Completeness can be viewed as a sophisticated sanity check for the Hoare logic, ensuring that one cannot get stuck in the verification because of some missing Hoare rules.

The state-space representation within Simpl is not fixed, rather it is a HOL type variable. It can be instantiated to meet the requirements and format of the program one attempts to verify. In Simpl all expressions are shallowly embedded, whereas compound statements are deeply embedded. The basic atomic statements however, are also modeled semantically as a state update function. This careful arrangement makes it possible to define and reason about the Hoare logic via the statement structure of the program and additionally provides the flexibility to instantiate the language with various state-space models or atomic operations that reflect the programming language under consideration.

To facilitate the usage of the Hoare logic within Isabelle/HOL, the application of the rules is automated as a verification condition generator. Moreover, an interface to software model checkers and termination analysis is provided [13].

3.2 Simpl to C0 Big-Step

We embed C0 into Simpl and use its verification environment to conduct C0 proofs. To transfer Simpl Hoare triples down to C0, we make sure that every behavior of the C0 program is also part of the Simpl counterpart and is thus covered by the Hoare triple. The C0 representation in HOL is a traditional deep embedding. First the abstract syntax of all relevant entities—types, values, expressions, and statements—is defined and then meaning is assigned to these entities, via a type system or the operational semantics. On the C0 syntax we define an abstraction to Simpl programs. Similarly, a C0 program state is abstracted to a corresponding Simpl state. The following key theorems allow us to transfer Hoare triples from Simpl to C0.

Theorem 3 (Simpl simulates C0). *Every execution of a C0 program is simulated by the execution of the corresponding Simpl program.*

Theorem 4 (Preservation of termination). *Termination of the Simpl program implies termination of the original C0 program.*

The proof of Theorem 3 is conducted by induction on the big-step execution of the C0 program (cf. [8, Chapter 8]). The most important invariant used for the induction is type safety. This matches the intuition of the abstraction that goes on between the C0 and the Simpl representation. In C0 typing is explicit whereas in Simpl it is mapped to HOL typing. To be more accurate Theorems 3 and 4

only hold for well-typed C0 programs and well-typed memory states. In C0 those constraints are encoded in predicates whereas in Simpl they are maintained by Isabelle's type-inference. Type-safe memory is also the prerequisite to justify the split heap representation in Simpl against the monolithic heap in C0.

A technical issue of the simulation proof stems from the state-space representation in Simpl. As we use a HOL record to encode program variables and heap components the HOL type of the state is not fixed for all C0 programs, but depends on the individual program. We achieve an individual HOL typing for the different variables and heap components, but cannot formulate Theorems 3 and 4 generically for all C0 programs within HOL. Instead we come up with a stratification of the theorems in two stages. A meta-theorem holds for all C0 programs and *assumes* commutation properties for the atomic state lookups and updates that appear in the program. For each individual C0 program we separately *discharge* these commutation properties. Fortunately, the second step is straightforward and can be automated. Hence, the resulting methodology is still generically applicable to all C0 programs. Isabelle's lightweight module concept of locales [14] supports this separation into a meta-theorem for all programs as well as the instantiation [15] for individual programs.

3.3 From Big-Step to Small-Step

The theorems to transfer Hoare triples from the big-step semantics down to the small-step semantics are analogous to Theorems 3 and 4.

Theorem 5 (Big-step simulates small-step). *Every terminating C0 small-step computation can be simulated by a big-step execution.*

Theorem 6 (Preservation of termination). *Termination of the C0 program in big-step semantics implies termination in the small-step semantics.*

Intuitively, we have to bring two orthogonal aspects of the semantics together: (i) computation: from a sequence of steps to a big-step and (ii) memory: from flat to aggregate values. We reflect this separation in the proofs by introducing an intermediate semantic level. A small-step semantics with the same memory model as the big-step semantics as well as the same (computational) granularity as the small-step semantics. First we focus on the computation aspect and prove that a terminating sequence of computation steps in the intermediate semantics can be simulated by a single big-step execution and that termination of the big-step semantics implies termination of the intermediate small-step semantics. In the second step we focus on the memory and define an abstraction function that maps a small-step configuration to a configuration in the intermediate semantics. We prove that every step in the small-step semantics can be simulated in the intermediate semantics, which also implies that termination is preserved.

Again, the simulation only holds for well-typed programs and well-typed memories. Since the memory representation varies in the different semantics the notion of a well-typed memory also differs. The low-level notion introduced in the small-step semantics together with the abstraction function to aggregate values is sufficient to imply the relevant restrictions at the big-step level.

4 Compiler Correctness

Software verification in Verisoft does not stop at the C0 level. To allow execution of verified programs on the 'real' hardware they are compiled to binary code. This translation could itself introduce errors into an otherwise verified C0 program. Thus, verification of the translation process is essential for pervasive systems verification if the system software and applications are implemented in a high-level programming language.

We close this gap by verifying a simple, non-optimizing compiler translating C0 to VAMP assembler code [16,17]. In addition to a *compiling specification* in Isabelle/HOL the compiler is *implemented* as a C0 program. Both the compiling specification and the compiler implementation have been formally verified. For the former we have proven a small-step simulation theorem stating that the original C0 program (executed by the C0 small-step semantics) and the compiled code behave equivalently. For the latter we have shown using our Hoare logic environment that it produces exactly the same list of assembler instructions as the compiling specification. Both results are combined into a single theorem. In this paper we can focus on the correctness of the compiling specification.

The correctness theorem presented below applies to *translatable programs*, which must be well-formed and fulfill certain resource restrictions of the target machine, e.g., to deal with limited memory size. Since the C0 small-step semantics and the VAMP assembler machine are deterministic this allows us to transfer Hoare triples down to the assembler and hardware layer.

The code generation algorithm of the C0 compiler follows directly the structure of the input program. It starts by iterating over all functions in the function table and generates code for their bodies. The code generation for statements and expressions—in the context of a certain function—is done by a simple recursive algorithm which follows the structure of the corresponding data types.

Essentially, the main theorem of the compiler correctness proof states that for all steps i of the C0 machine there exists a corresponding step number s_i such that after s_i steps the assembler machine is consistent with the C0 machine after i steps. This consistency is stated formally by a simulation relation between configurations of the C0 machine and configurations of the VAMP assembler machine. The simulation relation consists of several parts. *Control consistency* states that the VAMP's program counters point to the code of the first statement in the current C0 program rest. *Code consistency* requires that the compiled code of the C0 program remains intact, i.e., it forbids self-modification. *Value consistency* requires for all *reachable* variables g of basic type that the (C0) value of g is stored in the VAMP configuration at the allocated address of g. For reachable pointer variables p which point to some variable g we require that the value stored at the allocated address of p in the VAMP machine is the allocated base address of g. This defines a subgraph isomorphism between the reachable portions of the heaps of the C0 machine and the VAMP machine. *Stack consistency* is a technical predicate about the implementation of the run time stack and the content of some special registers.

Because the set of valid variables of a C0 machine changes with *new* statements, function calls, and returns, and because garbage collectors can change the allocated base address of variables on the heap, the simulation relation is parametrized by the current allocation function, which maps variables to their allocated base address in the target machine.

Theorem 7 (Compiling Specification Correctness). *Let p be a translatable C0 program, and assume that the initial assembler configuration holds the compiled code of p. Then, for all steps i of the C0 machine executing program p that did not reach an error state or produced a stack overflow there exists an assembler step number s_i and an allocation function $alloc_i$ such that the C0 machine after i steps (of small-step semantics) is consistent with the assembler machine after s_i steps with respect to $alloc_i$.*

5 Extended Hoare Logic

The language stack described in Sects. 3 and 4 allows us to transfer program properties from the Hoare logic down to the assembler level. However, in the context of system code we have to deal with portions of inline assembler code that break the abstraction of structured C0 programs: low-level entities (like the state of a device) may become visible even in the specification of code that is only a client to the inline assembler parts. To avoid doing all the verification in the lower semantic levels we extend the Hoare logic to represent the low-level actions on an abstract extension of the state-space. Inline assembler code is encapsulated in so-called *XCalls* at the Hoare logic level, which are modeled as atomic state updates. The correctness proofs of Sect. 3 allow us to transfer XCalls down to the assembler semantics, where they are finally discharged by an implementation proof. The compiler correctness proof only covers the assembler free portions of the code. Assembler code is inserted literally into the target code. Since compiler correctness is stated relatively to small-step semantics (of both the C0 and the assembler machine) the results can be extended to the combined computation: As long as the ordinary C0 code is executed the compiler correctness theorem covers the computation of the translated code and guarantees that we arrive at corresponding configurations at the machine and the C0 semantics level. Then the inline assembler code is executed according to the assembler semantics. The implementation proof (which also includes termination) ensures that its effect is the one expected by the abstract XCall semantics. Hence, at the end of the assembler computation we again arrive at corresponding final configurations. Then we can switch back to the compiler correctness result, and so on.

6 Dealing with Devices

Device drivers are an integral part of system software. Not only high-level functionality such as file I/O or networking depend on devices. Even basic operating system features, such as demand paging (Sect. 7), need correctly implemented

drivers. Hence, any verification approach of computer system stacks should deal with driver correctness. Nonetheless, when proving functional driver correctness it does not suffice to reason only about code running on a processor. Devices themselves and their interaction with the processor also have to be formalized.

We model devices as deterministic finite-state-machines communicating with both an external environment and the processor. The external environment is used to model non-determinism and communication; the network interface card, for example, sends and receives network packets. The processor accesses a device by reading or writing special address ranges. The devices, in turn, can signal interrupts to the processor; DMA is not considered. In Verisoft, we formalized models for an ATAPI disk, a serial interface, and an automotive bus controller.

At the gate-level hardware, devices are executed in lock-step. However, moving to the instruction set architecture, we loose granularity and hence timing information. We compensate for this loss by introducing interleaved execution of devices and the processor. An oracle, called *executing sequence*, determines when some device or the processor is allowed to take steps. In the following we refer to this interleaved semantics as the combined system.

Obviously, when proving correctness of a concrete driver, an interleaved semantics of all devices is extremely cumbersome. Integration of results into traditional Hoare logic proofs also becomes hardly manageable. Preferably, we would like to maintain a sequential programming model or at least, only bother with interleaved steps of those devices controlled by the driver we attempt to verify.

A basic observation of our overall model is that device and processor steps that do not interfere with each other can be swapped. For a processor and a device step, this is the case if the processor does not access the device and the device does not cause an interrupt. Similarly, we can swap steps of devices not communicating with each other. Utilizing this observation we reorder execution sequences into parts where the processor accesses no device or only one device. All interleaved and non-interfering device steps are moved to the end of the considered part and hence a (partially) sequential programming model is obtained.

Theorem 8 (Reordering of Non-Interfering Devices). *The interleaved execution of the combined system can be reordered into non-interleaved chunks of processor and device steps, such that the resulting execution simulates the original computation.*

Note that compiled, pure C0 programs never access devices, because data and code segments must not overlap with device addresses. Hence, all interleaved device steps can be delayed until some inline assembler statement is encountered. In combination with the concept of XCalls, lifting driver correctness statements to pure sequential Hoare logic, as demonstrated in the next section, becomes feasible. More generally, the execution of drivers controlling different (non-interfering) devices can also be separated, enabling modular verification of device drivers. The sketched reordering theory is developed and applied to the verification of a hard disk driver in [18]. This driver is part of the page-fault handler outlined in the next section, as it is used to swap pages between the physical memory and the hard disk.

7 Property Transfer Example: Page-Fault Handler

The formal verification of an academic operating system microkernel is a Verisoft subproject. The microkernel contains a verified page-fault handler that, in collaboration with other memory management routines, implements isolated, virtual memory for the user processes by means of demand paging [19]. Implemented in about 300 lines of C0 code with several calls to a hard disk driver written in assembler, it is a perfect candidate to illustrate our verification methodology.

Problem. One of the most challenging parts of verification of the Verisoft microkernel is memory virtualization, i.e., to ensure that each user process controls its own, large, and isolated memory. User processes access memory by virtual addresses, which are subsequently translated to physical ones. Modern computer systems implement virtual memory by demand paging: small consecutive chunks of data, called pages, are either stored in a fast but small physical memory or in a large but slower swap memory (usually a hard disk). The page table, a data structure both accessed by the processor and by software, maintains whether a page is in the swap or the physical memory. A process attempting to access a page currently in swap memory causes the processor to signal a page-fault interrupt. On the hardware side, the memory management unit (MMU) triggers the interrupt and translates from virtual to physical page addresses. On the software side, the *page-fault handler* reacts to page-faults by loading the requested page to the physical memory. If the physical memory is full, some other page is swapped out (cf. Fig. 2).

The verification objective is a simulation proof between a processor running a page-fault handler and user processes with virtual memory.

Implementation Model and Correctness. The correctness theorem is stated at the level of VAMP assembler combined with interleaved devices. One of the devices is instantiated with an ATA/ATAPI hard disk model. We call the processor model at the assembler level *physical machine*. It makes address translation and page-faults visible. User processes are modeled by *virtual machines* that, in essence, do not have address translation.

Theorem 9 (Virtual Memory Simulation). *The physical machine with a page-fault handler and a hard disk simulates virtual machines.*

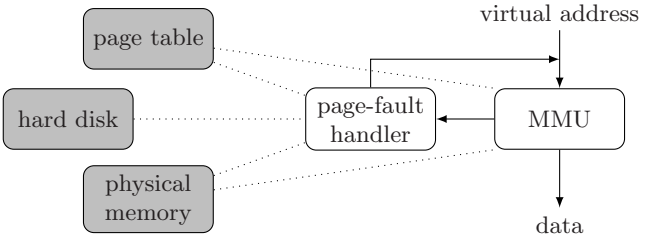


Fig. 2. Concept of Paging

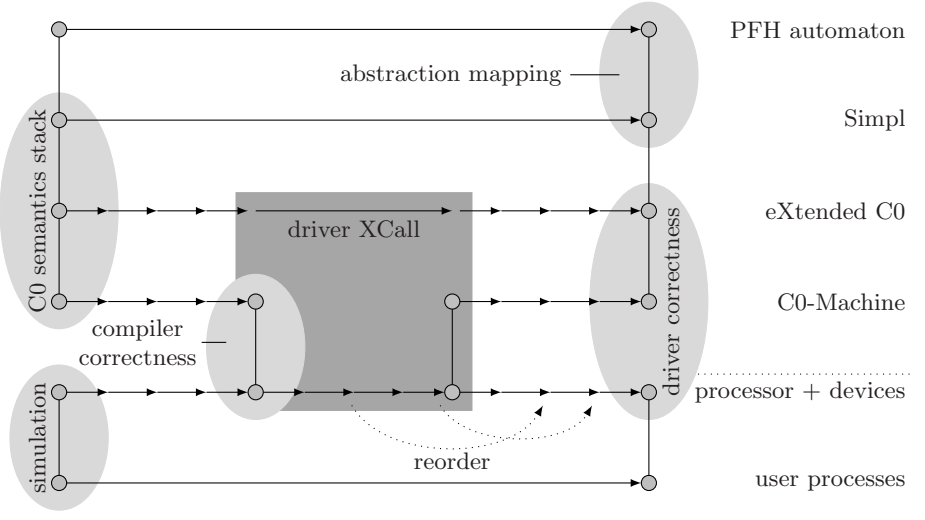


Fig. 3. Putting It All Together – Correctness of the Page-Fault Handler

A simulation relation projects the virtual memories of processes to the memory of the physical machine and the swap (i.e., sector) memory of the hard disk. Proving the correctness of memory virtualization is a step-to-n-steps simulation between the virtual and physical machine. The interesting case is the occurrence of a page-fault during the execution of a load or store instruction. In all other cases the semantics of the virtual and physical machines almost coincides. Thus, a proof of Theorem 9 boils down to justification of the next theorem.

Theorem 10 (Page-Fault Handler Correctness). *The simulation relation is preserved under an execution of the page-fault handler.*

The page-fault handler is a C0 program with calls to assembler subroutines, implementing the hard disk driver. The semantics of the assembler portions is encapsulated in XCalls. The extended state consists of a sector memory of the hard disk and the part of the physical memory that cannot be accessed by C0 variables. This enables us to verify the page-fault handler in our Hoare logic environment. We do not conduct all the verification even at this level. We introduce a higher-level concept for the data structures and algorithms of the page-fault handler to which we refer to as *PFH automaton*. For instance, a doubly linked list, a pointer data structure used by the page-fault handler implementation, is abstracted to a Isabelle/HOL list of references. This is formalized as an *abstraction mapping* and is proven to be preserved under page-fault handler executions.

Formal Verification. Conceptually, Theorem 10 follows from the page-fault handler functional correctness. The overall approach is sketched in Fig. 3. We first

specify and prove all necessary and sufficient properties in terms of the PFH automaton. By proving in Hoare logic that the abstraction mapping holds in the state after executing the page-fault handler provided it holds before, we obtain the desired properties at the level of Simpl. Applying Theorems 3 and 4 we map these results down to the C0 big-step semantics level and further via Theorems 5 and 6 to the level of the C0 small-step semantics. We justify the XCalls to the hard disk driver by plugging in the results from [18]: driver correctness can largely be shown in a sequential setting, ignoring other devices than the hard disk. Exploiting reordering (Theorem 8) we generalize this result to arbitrary interleaved sequences. Next, by the compiling specification correctness Theorem 7 we are able to state the page-fault handler functional correctness in terms of the assembler semantics. Altogether we conclude Theorem 10.

8 Conclusion: Proving as an Engineering Science?

In this article we presented an overview of the core layers in the Verisoft project. They support effective reasoning at proper abstraction levels while still allowing to transfer properties down to the machine level. Our approach is quite fundamental, since every layer and all the theorems are formalized within the theorem prover Isabelle/HOL. Integrating devices into the system stack in a way that allows to preserve sequential reasoning for major parts of the system can be regarded as major achievement of the Verisoft project. The example of memory virtualization via demand paging gives strong confidence in the appropriateness of the system stack and the verification methodology.

The meta-theorems that allow us to transfer results from one level to another only had to be proven once and for all. The effort for these proofs is compensated by the improved effectiveness in conducting major parts of the verification at a more abstract level. However, there is a difference between just having proven the transfer theorems (which gives an abstract notion of the soundness of the abstractions) and really instantiating them to transfer concrete properties. These instantiations can become quite tedious, since they demand to formulate the 'same' property at different abstraction levels where usually additional invariants have to be considered in order to exploit abstract information at the lower level.

All the Verisoft work done in Isabelle/HOL sums up to tens of megabytes of formal proof documents and marks the cutting edge of current academic verification projects, challenging the verification tools, in particular the theorem prover Isabelle (version 2005), as well as the social process to organize dozens of researchers at different places collaborating on the theories.

In the context of pervasive verification, employing automatic verification tools is a particular challenge: all tools need to be seamlessly integrated into the main proof tool Isabelle. We have done this for a small number of tools. At the hardware level a model checker has been integrated as an oracle [20]. Tactics have been implemented for verification condition generation and the per-program assumptions

of Theorems 3 and 4. Computational reflection was used to discharge various wellformedness constraints for property transfer between the C0 semantics. Although promising in general the integration of software model checking and termination analysis into the Hoare logic did not reach a mature state.

The final correctness theorem of the page-fault handler is the tip of an iceberg, importing various models, theorems, and proofs developed by different authors with different background and different style of formalization. This variety naturally results in friction losses, especially as the focus of this work was to show that one can 'get the verification done', even for a realistic system stack. These friction losses are annoying for the simpler proofs and tend to culminate around the more sophisticated arguments, making them even harder to obtain. For example the simulation of the C0 big-step and the intermediate small-step semantics took only about two weeks of work, since all the models were out of one hand and neatly fitted together. However, the simulation of the intermediate small-step semantics and the small-step semantics with flattened values took almost a year. This is due to some technically involved arguments and intermediate notions, especially regarding the memory update, but also due to a bunch of minor deviations in the formalization of corresponding aspects, which resulted from the fact that the theories were developed at different sites. Some were adapted and others were just bridged, because a change would have led to an enormous amount of work to accommodate the existing proofs.

An interesting social phenomenon is a kind of 'advice-resistance' and an extremely high tolerance level of the users. The lower the system level, the more invariants and assumptions appear in the theorems and proofs. These get both hard to grasp for the user and also ineffective to deal with by Isabelle's built in automation like the simplifier. To a large degree these issues can be avoided by switching from the tactical apply style to structured Isar proofs [21]. However, most users started working with apply-scripts (since the Isabelle tutorial [22] is still written in that style) and will not switch to the new paradigm until completing the proof at hand, even if that takes longer than expected.

At the ML level Isabelle provides means to profile the system. However, there is no support to effectively analyze and optimize a big theory corpus at the user level to answer questions like: Where are performance bottlenecks in the proofs? Do lemmas appear in the 'right' theory or library? Are some lemmas repeatedly proven? Moreover, Isabelle's built in lemma search assumes that theories are already loaded. This is rarely the case in large developments like ours, unless the vision of shared heaps or a proof wiki becomes reality.

One of the major challenges of Verisoft is the integration of various models, which technically means a high dependency on 'all' other theories. This lack of modularity leads to long turnaround cycles and a high sensitivity to changes of other users. Isabelle's theory structure relies on the user's discipline and does not necessarily impose proper boundaries or reflect the real dependencies. Using Isabelle's locales to disentangle theories may bring some relief, but still a more fine-grained management of the formal entities within Isabelle seems promising.

References

1. Bevier, W.R., Hunt Jr., W.A., Moore, J S., Young, W.D.: An approach to systems verification. Journal of Automated Reasoning 5(4), 411–428 (1989)
2. Moore, J S.: A grand challenge proposal for formal methods: A verified stack. In: Aichernig, B.K., Maibaum, T.S.E. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 161–172. Springer, Heidelberg (2003)
3. Neumann, P.G., Feiertag, R.J.: PSOS Revisited. In: 19th Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV, USA, pp. 208–216. IEEE Computer Society, Los Alamitos (2003), http://csdl.computer.org/comp/proceedings/acsac/2003/2041/00/20410208abs.htm
4. Walker, B.J., Kemmerer, R.A., Popek, G.J.: Specification and verification of the UCLA Unix security kernel. Comm. ACM 23(2), 118–131 (1980)
5. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: Taking microkernels to the next level. SIGOPS Oper. Syst. Rev. 41(4), 3–11 (2007)
6. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL 2007, pp. 97–108. ACM Press, New York (2007)
7. Tuch, H., Klein, G.: Verifying the L4 virtual memory subsystem. In: Proc. NICTA Formal Methods Workshop on Operating Systems Verification, pp. 73–97 (2004)
8. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (April 2006)
9. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 189–206. Springer, Heidelberg (2007)
10. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: The VFiasco project. In: SIGOPS 2002, pp. 165–169. ACM Press, New York (2002)
11. Tverdyshev, S., Shadrin, A.: Formal verification of gate-level computer systems. In: Rozier, K.Y. (ed.) LFM 2008. NASA STI, NASA, pp. 56–58 (2008)
12. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 7, pp. 23–50. Edinburgh University Press (1972)
13. Daum, M., Maus, S., Schirmer, N., Seghir, M.N.: Integration of a software model checker into Isabelle. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 381–395. Springer, Heidelberg (2005)
14. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 34–50. Springer, Heidelberg (2004)
15. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 31–43. Springer, Heidelberg (2006)
16. Petrova, E.: Verification of the C0 Compiler Implementation on the Source Code Level. PhD thesis, Saarland University, Computer Science Department (2007)
17. Leinenbach, D.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Computer Science Department (2008)
18. Alkassar, E., Hillebrand, M.A.: Formal functional verification of device drivers. In: Woodcock, J., Shankar, N. (eds.) VSTTE 2008. LNCS, vol. 5295. Springer, Heidelberg (2008)

19. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 109–123. Springer, Heidelberg (2008)
20. Tverdyshev, S., Alkassar, E.: Efficient bit-level model reductions for automated hardware verification. In: TIME 2008, pp. 164–172. IEEE Computer Society Press, Los Alamitos (2008)
21. Wenzel, M.: Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents. PhD thesis, Technische Universität München (2002)
22. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)

Formal Functional Verification of Device Drivers

Eyad Alkassar[1,⋆] and Mark A. Hillebrand[2,⋆]

[1] Saarland University, Saarbrücken, Germany
eyad@wjpserver.cs.uni-sb.de
[2] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
mah@dfki.de

Abstract. We report on the formal functional verification of a simple device driver for an ATAPI hard disk in Isabelle/HOL. The proof is based on a functional model of the hard disk, which has been integrated into the instruction set architecture of a verified RISC processor as one of several memory-mapped devices. The result is an interleaved computational model, in which the devices and the processor take turns in execution. Even in this concurrent context, the verification can be kept largely sequential and modular with respect to the other devices. This is made possible by sound reordering of computation traces, given that devices do not interfere with each other and the driver monopolizes the hard disk. To the best of our knowledge, this paper presents the first formal functional verification of a device driver against a realistic device and system model.

1 Introduction

The Verisoft project deals with formal pervasive verification, attempting the complete verification of several example computer systems [1]. The systems being considered employ I/O devices for non-volatile storage, communication, or user interaction. The devices are integrated at the hardware level as memory-mapped devices and controlled via device drivers running in system or user mode.

When pervasively verifying the correctness of such systems, a stack of formal computational models has to be built, reflecting the structure of the implementation; implementations in one layer must simulate the model of the next higher layer. All models in the stack must include formal models of devices. From one level to the next the representation of a certain device may change due to device drivers abstracting device behavior (consider, e.g., a hard disk versus a file system). The timing behavior of a device is usually not modeled exactly even in its most concrete representation and devices often interact with an external

⋆ Work funded by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems' and by the German Federal Ministry of Education and Research (BMBF) under grant 01 IS C38.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 225–239, 2008.

environment like a user or a network. Thus, the computational models in the stack have to be non-deterministic and concurrent.

The correctness statement of a device driver should be formulated with respect to the concurrent formal model it is implemented in. For the complete verification of the driver, not only this statement has to be shown but the correctness of all underlying device drivers as well. For user-level device drivers and for models in which devices are not accessible directly, typically several layers have to be considered, requiring simulation proofs between two or more concurrent computational models.

In this paper we consider the verification of a simple device driver for an ATAPI hard disk. We formulate and prove the correctness statement of the device driver in an assembly semantics with only a single device, the disk, visible. By reordering computation traces, we generalize this result to assembly computations involving other devices and lift it to the next-higher model, a C-like language providing device access only by calls to assembly drivers. Reordering exploits commutativity to sequentialize interleaved executions. To apply the theorems, different devices and drivers must be shown not to interfere with each other.

The theory and proofs above have been formalized in Isabelle/HOL [2].[1] To the best of our knowledge, this paper presents the first formal functional verification of a device driver against a realistic device and system model. The theorems of sound reordering of computation traces, which have been used in this proof, are useful for the verification of other device drivers as well and not specific to the considered verification example.

The remainder of this paper is structured as follows. In Sect. 2 we introduce device models and integrate them into larger computational models, i.e., here mainly an assembly semantics with devices. Assuming that devices are exclusively controlled by their respective drivers, we show in Sect. 3 how to transfer the correctness of a driver with respect to its device to the correctness of the driver in the full, running system. Moreover, we show that exclusive control of a device by its driver usually enables one to abstract from device and driver when going up one level in the stack of computational models. Thus, devices and their drivers have a simpler representation for client code. In Sect. 4 we present the device model of a hard disk, based on a subset of the ATAPI standard [3]. In Sect. 5 we present a simple hard disk driver for the disk, which writes a single page from memory to the hard disk. We sketch the formal correctness proof of the driver in a model with a single device. We then show how to generalize this results to the full system using the theory presented in Sect. 3. In Sects. 6 and 7 we present future work and conclude.

Related Work. Two earlier Verisoft publications are relevant for our work. We have reported on paper-and-pencil models and proofs related to a simple disk driver [4]. We extend this work in two important ways: models and proofs are formalized in Isabelle/HOL, and the models are now concurrent instead of lockstep. Thus, they are not restricted to disks, which are 'simple' for lack of external

[1] Theory files are available at `http://www.verisoft.de/VerisoftRepository.html`.

communication. In [5] we have reported on formal models of a serial interface and an architecture with devices, but not treated drivers. Here, we formally prove (disk) driver correctness up to the model of a high-level programming semantics.

So far most other device related verifications have either targeted the correctness of gate-level implementations or safety properties of drivers. In approaches of the former kind, simulation- and test based techniques are used to check for errors in the hardware designs. In particular, [6,7] deal with serial interfaces in that manner. In approaches of the latter kind the driver code is usually shown to guarantee certain API constraints of the operating system and hence cannot cause system crashes. For example, the SLAM project [8] provides tools for the validation of safety properties of drivers written in C. SLAM's success led to the deployment of the Static Driver Verifier (SDV) as part of the Windows Driver Foundation [9]. SDV automatically checks 65 safety rules concerning the Windows Driver API for device drivers. Hallgren *et al.* [10] modeled device interfaces for a simple operating system written in Haskell. Three memory-mapped I/O calls were specified: read, write, and test for valid region. However, the only correctness property being stated is the disjointness of the device address spaces.

In contrast to all mentioned approaches, we aim at the formalization and *functional* verification of drivers interacting with a device. Thus, it is not sufficient to argue about the device or programming model alone. Even in other ongoing systems verification projects, the L4.verified project [11] and the FLINT project [12], device behavior and driver correctness are not considered. To our knowledge, the only work similar in scope is the challenge proposed by Holzmann [13] dealing with the formal verification of a file system for a Flash device. In response to the challenge, Woodcock reports on the partial specification of the file system (the 'file store') and a refinement proof mapping the store to a Java program [14]. Simultaneously, the Flash hardware is being formalized [15]. Verifying a low-level Flash driver and integrating it into the filesystem proofs are future work. Concurrency is not an issue since only a single device is considered.

Reordering execution sequences to obtain atomic specifications have been well studied in literature under the topic of *reduction theorems*. Lipton proved safety properties of pre-/ post-condition style sequentially and propagated these to the implementation [16]. Cohen and Lamport extended this to liveness and a more fine-grained analysis of the reordered parts of the sequence [17,18]. Most reduction theorems assume that the implementation fulfills some non interference theorems. In contrast we prove this assumption on the atomic specification by exploiting a similar insight as reported in [19]. Justified by the memory mapped I/O architecture, the theory presented here is a specialization, enabling us to formulate even stronger reduction theorems than reported in literature.

2 Device Models and Models with Devices

Reasoning about driver correctness requires a detailed programming model. In our case, the driver is executed as an assembly program on a RISC instruction set architecture (ISA). A formal transition system of the ISA is first defined and its interface to devices is described. We proceed with an abstract device

model suitable for the modeling of memory-mapped devices; currently, we do not support device direct memory access (DMA). By combining the previous models we obtain a model where the processor and several devices run concurrently. We model this by introducing an oracle input called *event*, which determines whether some device or the processor takes the next step.

The concurrent model also serves as the specification of a concrete gate-level implementation of a processor with devices, which is an accurate model of the hardware. A simulation proof between these two models is presented in [20].

Processor Model. The processor model is the sequential programming model of the hardware as seen by a system software programmer. Machine configurations are 5-tuples $c_{\mathrm{P}} = (pc, dpc, gpr, spr, m)$ with the following components: the normal and the delayed program counters $c_{\mathrm{P}}.pc$ and $c_{\mathrm{P}}.dpc$ used to implement delayed branch, the general purpose register file $c_{\mathrm{P}}.gpr$, the special purpose register file $c_{\mathrm{P}}.spr$, and the byte addressable physical memory $c_{\mathrm{P}}.m$. We denote d consecutive memory cells starting at address a by $m_d[a] = (m[a+d-1], \ldots, m[a])$.

We support up to eight devices identified by natural numbers $i \in \{0, \ldots, 7\}$. These are mapped into the processor's memory at address ranges DA_i, which are mutually disjoint. Processor and devices may interact by (i) devices generating interrupts via external event lines $eev[i]$ or (ii) the processor accessing device ports at addresses in DA_i via regular memory instructions.

The interface for the latter operation is defined using two types. The processor requests a device access via the *memory interface input* and receives the device's response via the *memory interface output*; this naming convention is from the point of view of the devices.

Formally, let DA denote the union of all device addresses, let the predicates $lw(c_{\mathrm{P}})$ and $sw(c_{\mathrm{P}})$ indicate load and store word instructions, and let the functions $ea(c_{\mathrm{P}})$ and $RD(c_{\mathrm{P}})$ denote the memory and register operand addresses for such instructions. The memory interface input *mifi* is a quadruple: (i) the read flag $mifi.rd = lw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a load from a device port, (ii) the write flag $mifi.wr = sw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a store to a device port, (iii) the address $mifi.a = ea(c_{\mathrm{P}})$ is set to the effective address, which encodes the accessed device i in bits 12 to 14 and the accessed port in bits 2 to 11 (we support up to 1024 ports of 32 bit width per device), and finally (iv) the data input $mifi.din = c_{\mathrm{P}}.gpr[RD(c_{\mathrm{P}})]$ is set to the store operand.

The memory interface output *mifo* is a 32 bit response to a device port read.

The processor's ISA is formally defined by the output function ω_{P} and the transition function δ_{P}. The former takes a processor state c_{P} and computes a memory interface input *mifi*, cf. above. The transition function takes a processor state c_{P}, a device output *mifo*, and the devices' external event lines $eev[i]$, which indicate interrupts. It returns the next state of the processor c'_{P}. If all device interrupts are disabled in software and no device is accessed, the external event lines and the memory interface output are ignored. For such steps we use δ_{P} in an overloaded, unary variant, which operates on c_{P} only.

Devices. Devices are modeled as finite state transition systems interacting with the processor and with an external environment (e.g., a user or a network). In the

following let X denote a specific kind / type of device. The transition function δ_X takes a device state, an input from the external environment $eifi_X$, and an input from the processor *mifi*. It returns the next state, an output to the processor *mifo* and an output to the external environment $eifo_X$. Interrupts are signaled by a predicate ω_X over the device state.

In the models considered here a device either consumes an external or a processor input, never both simultaneously. Hence, in a step either *eifi* or *mifi* is 'empty', denoted with $eifi_\epsilon$ and $mifi_\epsilon$.

Combined System. In the overall system we study a model of one processor connected to several devices. A configuration c_{PD} of the combined system, which we also call global configuration, consists of a processor configuration $c_{\mathrm{PD}}.c_{\mathrm{P}}$ and a mapping $c_{\mathrm{PD}}.c_{\mathrm{D}}$ from device identifiers to device configurations.

The transition function δ_{PD} of the combined system has to distinguish whether the processor or a device executes next. Hence, it takes the current global configuration and an oracle input *ev* called event. The event equals P in case of a processor step or is a pair $(i, eifi)$ of device identifier and environment input in case of a device step. The transition function returns the next global configuration and an output *eifo* to the environment.

Let the function *da* indicate whether the processor wants to perform a local step or access a specific device. Formally, $da(c_{\mathrm{P}}) = i$ if $(sw(c_{\mathrm{P}}) \vee lw(c_{\mathrm{P}})) \wedge ea(c_{\mathrm{P}}) \in DA_i$ and $da(c_{\mathrm{P}}) = P$ otherwise.

In the definition of the transition function we distinguish three cases:

1. A *processor-device transition* is taken if it is the processor's turn, $ev = P$, and the current instruction accesses a device $da(c_{\mathrm{PD}}.c_{\mathrm{P}}) = i$ with type X. The device takes a step with δ_X, consuming the output $\omega_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}})$ of the processor and an empty external input $eifi_\epsilon$. For a read, the device returns an output *mifo* to the processor. The processor configuration is updated by applying δ_{P} to the current processor configuration, the memory output *mifo*, and the external event bit vector *eev*, defined as $eev[j] = \omega_X(c'_{\mathrm{PD}}.c_{\mathrm{D}}(j))$ for each device j of type X.

2. A *local processor transition* is taken if it is the processor's turn, $ev = P$, and the current instruction does not access a device, $da(c_{\mathrm{P}}) = P$. The processor configuration is updated by applying δ_{P} to the current processor configuration $c_{\mathrm{PD}}.c_{\mathrm{P}}$, a (dummy) device input, and the external event vector as defined above.

3. An *external device transition* is taken if there is an external input *eifi* for a device i of type X, i.e., $ev = (i, eifi)$. Only the configuration of device i is updated by applying δ_X with the processor input set to $mifi_\epsilon$.

Devices and the processor are executed in an interleaved way. A model run is defined by the start configuration and an *execution sequence* denoted by *seq*. The latter returns for a given step number t the oracle event input $seq(t) = ev$, i.e., it resolves the non-determinism.

The function Δ_{PD} is used to model a computation of the overall system. It takes a global start configuration, an execution sequence and a step number t as inputs. It returns a pair, the global configuration reached after applying the

transition function δ_{PD} for t times and the sequence of external output generated during this process.

When proving a property of the combined system, not all execution sequences have to be considered. For example, termination of drivers can typically only be shown if processor and devices are scheduled infinitely often, which must be guaranteed by the hardware implementation [20]. Hence, we define valid execution sequences as:

$$Seq_{\mathrm{V}}(seq) \equiv (\forall t\,.\,\exists k > t\,.\,seq(k) = P) \wedge (\forall t, i\,.\,\exists k > t, eifi\,.\,seq(k) = (i, eifi))$$

Correctness of drivers could depend on further device-specific restrictions of the environment. For example, for the hard disk the environment eventually signals termination of a read or write operation. Such assumptions are also formulated in terms of Seq_{V} and proven for the gate-level implementation.

3 Reordering and Abstraction

Obviously, when proving correctness of a concrete driver for a specific device, an interleaved semantics of all devices is cumbersome. Preferably, for the proof we would like to use a simpler programming model first, e.g., a sequential model or a model with just a single device, and then generalize the result. In this section we develop theory for that purpose.

We assume that we have driver code that exclusively controls a certain device X and only that device. For simplicity, in this section we use X both to identify the kind of the controlled device and its number $X \in \{0, \ldots, 7\}$. We also assume that all interrupts are masked in hardware via the special-purpose status register while the driver runs. In our scenario this restriction is not severe. On the one hand, interrupts of device X are assumed already being delivered to our driver. On the other hand, interrupts for other devices should be handled by different drivers in a manner transparent to the driver under verification. Thus, this problem is orthogonal to the one that we focus on here. Techniques for the verification of concurrent (assembly) programs apply in this case (cf. [21]).

A key observation in our scenario is that some steps in the computation of the combined model can be swapped without changing the outcome. This reordering is sound if devices do not influence each other. Swapping steps repeatedly, an execution of the driver in the combined model can be separated into steps involving only the driver and the controlled device followed by steps involving only other devices. Thus, correctness of the driver can be shown in a model with only the processor and the controlled device. Still, this model is concurrent. Two further simplifications may be applicable for parts of the driver execution. First, for phases not involving device access at all properties can be proven relative to just the isolated processor model. Second, for phases in which the device is in a stable state (by which we mean it does not react to external input) properties can be proven relative to a model without (external) device steps.

A similar technique can be applied for higher-level models with devices. For example, in Verisoft the bulk of all software is implemented in a type-safe fragment of C. Most of this code is verified in a Hoare logic verification environment

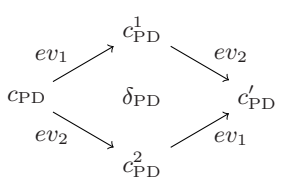


Fig. 1. Swapping Two Non-Interfering Steps

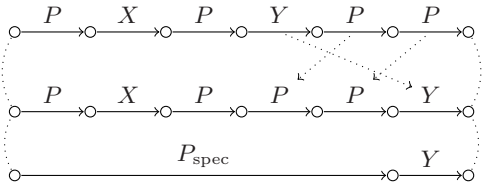


Fig. 2. Reordering and Abstraction of an Execution Sequence

for C. Integration of concurrent correctness results into traditional Hoare logic proofs is hardly manageable. It is much more convenient to show the correctness of high-level code against a sequential specification in which the driver calls are executed atomically. Because we use type-safe C we cannot allow direct access to device ports, which do not behave like regular memory. Hence, reordering techniques can be used for a concurrent C semantics with devices, separating C steps from device and driver execution steps.

A Basic Observation. A basic observation of our overall model is that device and processor steps not interfering with each other can be swapped. Assuming that interrupts are disabled, we say that two steps do not interfere if at least one is not a processor step and if they do not involve the same device; we call a device involved in a step if it is accessed by the processor or makes a step itself.

Recall that for a processor configuration c_{P} the function da indicates whether the processor makes a local step, $da(c_{\mathrm{P}}) = P$, or accesses a specific device, $da(c_{\mathrm{P}}) \in \{0, \ldots, 7\}$. For an event ev, we let $Da(c_{\mathrm{P}}, ev)$ denote the *set* of components involved in a step. We have $P \in Da(c_{\mathrm{P}}, ev)$ iff $ev = P$ and $X \in Da(c_{\mathrm{P}}, ev)$ iff $ev = (X, \ldots)$ or $da(c_{\mathrm{P}}) = X$.

For a global configuration c_{PD}, two events ev_1 and ev_2 with $Da(c_{\mathrm{PD}}.c_{\mathrm{P}}, ev_1) \cap Da(c_{\mathrm{PD}}.c_{\mathrm{P}}, ev_2) = \emptyset$ can be executed in arbitrary order, as depicted in Fig. 1:

$$\delta_{\mathrm{PD}}(\delta_{\mathrm{PD}}(c_{\mathrm{PD}}, ev_1)), ev_2) = \delta_{\mathrm{PD}}(\delta_{\mathrm{PD}}(c_{\mathrm{PD}}, ev_2), ev_1) \tag{1}$$

Lifting this observation to execution sequences is simple: we only have to ensure that a valid sequence remains valid after swapping. This is true since we restricted validity only to liveness. However, more complex assumptions over the environment could link input and output behavior of different devices or their relative speed. In this case the invariance of validity has to be proven before applying the reordering theorem that we present in the next section.

We define the following simple criterion to determine whether the basic observation holds over a set of execution sequences. Let $\pi(seq, i)$ denote the *projection* of a sequence seq to a given device i, i.e., it returns the subsequence of external steps of device i. A predicate over execution sequences is called separable if it can be expressed as a conjunction of predicates over projected execution sequences:

$$separable(Q) \equiv \exists p_0 \ldots p_7 \,.\, \forall seq \,.\, Q(seq) = p_0(\pi(seq, 0)) \wedge \cdots \wedge p_7(\pi(seq, 7))$$

Lemma 1 (Separable Valid Sequences). *If the valid sequence predicate is separable, then observation (1) holds over this set.*

Reordering. We study when sequential proofs over a given assembly code can be generalized to arbitrary computations. We abbreviate the configurations of a processor-local computation by c_{P}^t with $c_{\mathrm{P}}^{t+1} = \delta_{\mathrm{P}}(c_{\mathrm{P}}^t)$ and the configurations of a computation of the combined system by $c_{\mathrm{PD}}^{seq,t} = \Delta_{\mathrm{PD}}(c_{\mathrm{PD}}, seq, t)$.

In the simplest case the processor does not access any devices. Since interrupts are masked, processor computations yield the same result in the combined model regardless of device steps.

Lemma 2 (No Device Access)

$$(\forall t \leq T \,.\, da(c_{\mathrm{P}}^t) = P) \implies \forall seq \,.\, (c_{\mathrm{P}}^0 = c_{\mathrm{PD}}^{seq,0}.c_{\mathrm{P}} \implies \exists T' \,.\, c_{\mathrm{PD}}^{seq,T'}.c_{\mathrm{P}} = c_{\mathrm{P}}^T)$$

The stated lemma is useful in two situations. First, it allows to reason locally about local steps in the execution of a device driver. Second, it is applicable when reasoning about code of a high-level programming language without direct device access. In this case, code correctness proofs of the code in the high-level language can be performed purely sequentially.

In a more general case the processor accesses only a certain device X. We call such parts of the computation *pure*. This is our assumption for drivers; it can usually be shown statically or for local processor computations. Furthermore, we define device configurations to be *stable* if they do not change under external transitions. These predicates are defined formally as follows:

$$\begin{aligned} pure(c_{\mathrm{PD}}^0, seq, T, X) &\equiv \forall t < T \,.\, da(c_{\mathrm{PD}}^{seq,t}.c_{\mathrm{P}}) \in \{P, X\} \\ stable(c_X) &\equiv \forall eifi \,.\, \delta_X(c_X, eifi, mifi_\epsilon) = c_X \end{aligned}$$

The *empty sequence* is the schedule where only the processor takes steps. It is defined as $emp(t) = P$ for all t. In pure computations where the accessed device is stable, sequential properties proven over the empty sequence can be generalized to properties over arbitrary sequences.

Lemma 3 (Pure Sequences and Stable Devices)

$$\begin{aligned} &pure(c_{\mathrm{PD}}^0, emp, T, X) \wedge \forall t < T \,.\, stable(c_{\mathrm{PD}}^{emp,t}.c_{\mathrm{D}}(X)) \implies \\ &\qquad \forall seq \,.\, \exists T' \,.\, c_{\mathrm{PD}}^{emp,T}.c_{\mathrm{P}} = c_{\mathrm{PD}}^{seq,T'}.c_{\mathrm{P}} \wedge c_{\mathrm{PD}}^{emp,T}.c_{\mathrm{D}}(X) = c_{\mathrm{PD}}^{seq,T'}.c_{\mathrm{D}}(X) \end{aligned}$$

Note that stability of a device is a relatively strong assumption, but sufficient for handling a hard disk driver. For other devices, the notion of stability should be refined, requiring stability only for those parts of the device that are accessed by the processor.

In general, of course, driver correctness can not be shown solely using Lemmas 2 and 3. In the situations not covered by these lemmas, we may still assume

that only the processor or the device X are being scheduled. We call such fragments of an execution sequence *reduced*. Formally, we define

$$\mathit{reduced}(\mathit{seq}, T_1, T_2, X) \equiv \forall T_1 \leq t < T_2 \,.\, \mathit{seq}(t) = P \vee \mathit{seq}(t) = (X, \dots) \ .$$

Complementary, a fragment is free of steps of a device X or the processor iff

$$\mathit{free}(\mathit{seq}, T_1, T_2, X) = \forall T_1 \leq t < T_2 \,.\, \mathit{seq}(t) \neq P \wedge \mathit{seq}(t) \neq (X, \dots) \ .$$

If we have a separable valid sequence, the theorem below states that a pure computation can always be reordered into a reduced part and followed by a free part. The resulting overall state of both computations are equal.

Theorem 1 (Reordering of Sequences)

$$\begin{aligned} &\mathit{pure}(c^0_{\mathrm{PD}}, \mathit{seq}, T, X) \implies \\ &\exists \mathit{seq}', T_1, T_2 \,.\, \mathit{reduced}(\mathit{seq}', 0, T_1, X) \wedge \mathit{free}(\mathit{seq}', T_1, T_2, X) \wedge c^{\mathit{seq},T}_{\mathrm{PD}} = c^{\mathit{seq}',T_2}_{\mathrm{PD}} \end{aligned}$$

This theorem can be proven by repeatedly applying the basic observation above. Generalizing this result, the execution of drivers controlling different devices can also be separated, enabling modular verification of device drivers.

In Fig. 2 on page 231 we show an example of a complete execution of a driver for some device X. By applying Theorem 1 we soundly reorder the execution of any device Y after the termination of the driver (top line to middle line). The interaction between the driver and the corresponding device can now be specified by a single atomic state update (middle line to bottom line).

Abstraction. We examine the scenario that a program implemented in a high-level language wants to access devices by calling assembly drivers for these devices. We assume that the language does not provide direct device access, which is true for Verisoft.

To reason about correctness in this scenario we have to consider the compiled high-level program linked with the assembly driver in the model of the combined system (cf. Sect. 2). The compiler guarantees that the compiled code does not access devices by placing code and data region in regular memory (i.e., not on device ports). Whenever we execute a certain fragment of compiled code we can thus apply Lemma 2 and get to a processor configuration in the combined model that is equal to the processor configuration in the sequential processor model. Therefore, compiler correctness is preserved in a model with devices. In other words, device steps do not interfere with compiler correctness. Moreover, whenever we execute a certain fragment of the compiled code, we can shift the device steps beyond this fragment using Theorem 1 for the special case that there is no device access. In other words, high-level language and compiled code do not interfere with devices (and drivers).

Suppose a driver for device X is called by the high-level program. We apply Theorem 1 on the driver execution, obtaining a reduced fragment for the driver and its controlled device. This division not only eases the driver verification, but

also enables an atomic specification of the driver grouping involved processor and device steps to one semantical call (see the bottom line in Fig. 2). At the end of this fragment the postcondition of the driver ranging only over the device and the processor state can be established. All interleaved and non-interfering device steps are moved beyond the fragment and hence a (partially) sequential programming model is obtained. As a consequence, traditional program logics becomes also applicable to high-level programs including calls to that driver.

We have formally instantiated the described abstraction technique for the Verisoft C compiler in Isabelle/HOL and applied it to the hard disk driver correctness (cf. Sect. 5). Interrupts can be handled similarly if driver execution is transparent to / separated from high-level execution.

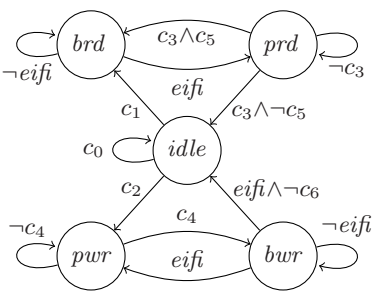
4 Hard Disk Model

Our formal hard disk controller model is based on a subset of the ATAPI standard [3]. We restrict ourselves to a few ATAPI commands and assume that only a single disk is hooked up to the controller, the master disk. Hence, we use the terms disk and controller interchangeably. Below, we sketch the definitions of hard disk state and operation, omitting many details due to space restrictions.

Hard disk state is modeled as a record c_{hd}. Hard disk operation is defined via a transition function δ_{hd}. It takes an external input *eifi*, a memory interface input *mifi*, and a current configuration c_{hd}. It returns an updated configuration c'_{hd}, a memory interface output *mifo*, and an external output *eifo*. In Sect. 2 we have already defined the signature of the memory interface. The external interface for the disk is quite simple. The disk does not produce an external output *eifo*; we will omit it from now on. The external input $\mathit{eifi} \in \{0,1\}$, also known as trigger, indicates when the disk completes certain operations; we abstract from exact timing. For liveness reasons, the trigger must be active infinitely often. This is a (separable) environment restriction we need to make, cf. Sect. 3.

We now describe the hard disk state in more detail. Hard disks are parameterized over the number of sectors $0 < c_{\mathrm{hd}}.S \leq 2^{28}$ they store. Each sector stores 128 words, making up for a maximum content of $c_{\mathrm{hd}}.S \cdot 2^9 \leq 128\,\mathrm{GB}$,

The disk is accessed by issuing commands to it. We only model three commands: *reset* initializes the disk state; *read* and *write* load and store a range of sectors. This range is processed sector by sector. During command execution, the sector range that remains to be processed is identified by a start sector $c_{\mathrm{hd}}.\mathit{lba} \in \mathbb{N}_{<c_{\mathrm{hd}}.S}$ and a sector counter $c_{\mathrm{hd}}.\mathit{scnt} \in \mathbb{N}_{<257}$. Each sector is first being transferred into an internal (volatile) buffer of the disk and then to the processor resp. the disk. The internal buffer is represented as a mapping $c_{\mathrm{hd}}.\mathit{buf} : \mathbb{N}_{<128} \to \mathbb{N}_{<2^{32}}$. The processor accesses this buffer sequentially by reading or writing the *data port* of the disk; each such access increments the buffer pointer $c_{\mathrm{hd}}.\mathit{bp} \in \mathbb{N}_{<128}$ that serves as an index into the internal buffer. Read and write commands can be executed in two modes. In *polling mode*, the processor queries the disk for the completion of a sector transfer; in *interrupt mode*, the disk causes an interrupt for each sector transfer. The interrupt enable



$c_0 = \neg wr(mifi, cmd_{\mathrm{p}})$
$\hat{c} = c_{\mathrm{hd}}.scnt \neq 0 \wedge (c_{\mathrm{hd}}.S \geq c_{\mathrm{hd}}.lba + c_{\mathrm{hd}}.scnt)$
$c_1 = cmd(mifi, rd_{\mathrm{c}}) \wedge \hat{c}$
$c_2 = cmd(mifi, wr_{\mathrm{c}}) \wedge \hat{c}$
$\tilde{c} = (c_{\mathrm{hd}}.bp = 127)$
$c_3 = rd(mifi, data_{\mathrm{p}}) \wedge \tilde{c}$
$c_4 = wr(mifi, data_{\mathrm{p}}) \wedge \tilde{c}$
$c_5 = c_{\mathrm{hd}}.scnt > 0$
$c_6 = c_{\mathrm{hd}}.scnt > 1$

Fig. 3. Regular State Transitions

flag $c_{\mathrm{hd}}.ien \in \{0,1\}$ indicates the current mode. It is zero for polling mode. In interrupt mode, the pending interrupt flag $c_{\mathrm{hd}}.pint \in \{0,1\}$ indicates whether a hard disk interrupt waits to be serviced by the processor. The interrupt predicate thus simply equals the pending interrupt flag, i.e., $\omega_{\mathrm{hd}}(c_{\mathrm{hd}}) = c_{\mathrm{hd}}.pint$.

All of the processor-device interaction we sketched above (other than the disk generating interrupts) takes place by the processor accessing the hard disk's eight ports. The sector count port $scnt_{\mathrm{p}}$ defines the number of sectors to process. The sector number, cylinder low, cylinder high, and drive head ports ($snumb_{\mathrm{p}}$, $cyll_{\mathrm{p}}$, $cylh_{\mathrm{p}}$, and $drvhd_{\mathrm{p}}$) define the 28-bit start sector represented as a quadruple of $3 \cdot 8 + 4$ bits. The device control port $devcntrl_{\mathrm{p}}$ selects polling or interrupt mode. The command port cmd_{p} is used to issue commands when written and check polling status when read. Finally, the internal buffer is accessed via the data port $data_{\mathrm{p}}$. The data port has width 32 bits, all other ports have width 8 bits.

Hard disk operation is governed by a small control automaton with state $c_{\mathrm{hd}}.cs \in \{idle, brd, bwr, prd, pwr, err\}$. In *idle* state, the disk awaits new commands. Read commands will loop over states *brd* and *prd*. In the former the disk fills its buffer, which is then read out by the processor. Likewise, write commands loop over states *pwr* and *bwr*. In the former the processor fills the disk's buffer, which is then written by the disk. The state *err* is an error state.

We call non-reset, non-error state transitions of the control automaton *regular*. These transitions are shown in Fig. 3. Edges are labeled with transition constraints, where $rd(mifi, x) = mifi.rd \wedge (mifi.a = x)$ and $wr(mifi, x) = mifi.wr \wedge (mifi.a = x)$ indicate read and write accesses to port x and $cmd(mifi, c) = wr(mifi, cmd_{\mathrm{p}}) \wedge (mifi.din = c)$ indicates the issuing of command c.

Let us outline the major distinctions in the transition function (in addition to the control state transitions). We assume $\delta_{\mathrm{hd}}(eifi, mifi, c_{\mathrm{hd}}) = (c'_{\mathrm{hd}}, mifo)$.

Issuing the reset command, $cmd(mifi, rst_{\mathrm{c}})$, has top priority: the disk control enters the idle state, $c'_{\mathrm{hd}}.cs = idle$, and the buffer pointer, the interrupt enable flag, and the pending interrupt flag are set to zero. The other components do not change; the value of *mifo* is irrelevant (for any processor write, in fact).

If no reset command is issued an error transition may be taken, $c'_{\mathrm{hd}}.cs = err$. Absence of error transitions should guarantee that the processor handles the device correctly. In addition to obvious error transitions, e.g., write to a read-only

port, we also use error transitions for modeling shortcomings, e.g., an attempt to issue an unmodeled command. We do not define the error conditions here.

Finally, we distinguish two types of regular transitions, which do not occur simultaneously. Regular, processor-initiated transitions are used to set up command parameters, start commands, access the internal buffer, or query the disk status. The definition of the transition function for these cases is easy. As side effects, the buffer pointer is incremented for a data port access and the pending interrupt flag is cleared for a disk status check.

Regular, external transitions are initiated by the external trigger flag *eifi*, which only has an effect in states *bwr* or *brd*, waiting for a sector to be written to or read from the disk. Such a transition has side effects. The start sector is incremented and the sector count is decremented, $c'_{\mathrm{hd}}.lba = c_{\mathrm{hd}}.lba + 1$ and $c'_{\mathrm{hd}}.scnt = c_{\mathrm{hd}}.scnt - 1$. For buffer write, the buffer is copied to the disk, $c'_{\mathrm{hd}}.sm_{128}(c_{\mathrm{hd}}.lba \circ 0^9) = c_{\mathrm{hd}}.buf$. For buffer read, $c'_{\mathrm{hd}}.buf = c_{\mathrm{hd}}.sm_{128}(c_{\mathrm{hd}}.lba \circ 0^9)$. The pending interrupt flag is turned on in interrupt mode, $c'_{\mathrm{hd}}.pint = c_{\mathrm{hd}}.ien$.

5 Hard Disk Driver and Correctness

We present a simple assembly device driver for which we have formally proven correctness in Isabelle/HOL [2] based on the combined system from Sect. 2 and the theory developed in Sect. 3. The driver writes a 4 K page (8 sectors) from the processor's memory, starting at address a, to the disk, starting at sector b. Its code is shown in Fig. 4. We use MIPS-like syntax; GPRs are written as `r`k, memory operands as $imm(RS1)$. Arrows indicate jump targets; according to the delayed PC, instructions in delay slots are always executed.

The code can be structured into five main parts. In part 0, we set up all parameters for the disk write command in the registers. For example, the start sector index b is decomposed into the sector number, cylinder low, cylinder high, and drive index. In part 1, command parameters are written to the disk's configuration ports. Interrupt mode is disabled in step (1.2) and the write command is issued in step (1.8). Each iteration of the outer loop in steps (2.1) to (5.3) copies one sector from the main memory of the processor to the sector memory of the disk. One sector consists of 128 words. The first inner loop copies word

```
andi r16,r15,#255    (0.1)
srli r17,r15,#8      (0.2)
andi r17,r17,#255    (0.3)
srli r18,r15,#16     (0.4)
andi r18,r18,#255    (0.5)
srli r19,r15,#24     (0.6)
andi r19,r19,#15     (0.7)
addi r19,r19,#224    (0.8)
addi r3,r0,#wr_c     (0.9)
addi r4,r0,#2       (0.10)
addi r12,r0,#8      (0.11)

xori r1,r0,#Da(X_hd) (1.1)
sw devctrl_p(r1),r4  (1.2)
sw scnt_p(r1),r12    (1.3)
sw snumb_p(r1),r16   (1.4)
sw cyll_p(r1),r17    (1.5)
sw cylh_p(r1),r18    (1.6)
sw drvhd_p(r1),r19   (1.7)
sw cmd_p(r1),r3      (1.8)

→addi r10,r0,#128    (2.1)
 →lw r3,0(r2)        (3.1)
  sw data_p(r1),r3   (3.2)
  subi r10,r10,#1    (3.3)
  bnez r10,#-16      (3.4)
 addi r2,r2,#4       (3.5)
→lw r3,stat_p(r1)    (4.1)
 sgei r14,r3,#128    (4.2)
 bnez r14,#-12       (4.3)
nop                  (4.4)
 subi r12,r12,#1     (5.1)
 bnez r12,#-48       (5.2)
nop                  (5.3)
```

Fig. 4. Device Driver

after word from the processor's memory to the internal disk buffer. When the buffer is full (i.e., after one complete sector) the driver enters the second inner loop, which polls until the hard disk has written its buffer.

In the following we will abbreviate component access to the hard disk and the processor with $c_{\mathrm{hd}}^{seq,t} = c_{\mathrm{PD}}^{seq,t}.c_{\mathrm{D}}(X_{\mathrm{hd}})$ and $c_{\mathrm{P}}^{seq,t} = c_{\mathrm{PD}}^{seq,t}.c_{\mathrm{P}}$ respectively.

The correctness of the driver is proven for all valid execution sequences. It states that when driver execution terminates, the page located in the processor memory is finally copied to the sector memory of the disk. Furthermore the physical memory of the processor stays unchanged.

Theorem 2 (Hard Disk Driver Correctness). *Assume memory address and start sector are in the first two registers,* $c_{\mathrm{P}}^{0}.gpr[1] = a$ *and* $c_{\mathrm{P}}^{0}.gpr[2] = b$*. The hard disk is assumed to be idle,* $c_{\mathrm{hd}}^{0}.cs = idle$*. Then it holds:*

$$\forall seq\,.\,Seq_{\mathrm{V}}(seq) \implies \exists t\,.\,c_{\mathrm{hd}}^{seq,t}.sm_{8\cdot 128}[b\cdot 128] = c_{\mathrm{P}}^{0}.m_{4\cdot 8\cdot 128}[a] \wedge c_{\mathrm{P}}^{seq,t}.m = c_{\mathrm{P}}^{0}.m$$

In the following we apply the theory developed in Sect. 3 for hard disk driver correctness. Since the first part of the code does not access any device at all, with Lemma 2 we can prove its correctness resorting only to ISA semantics. Next we establish that during part 1 only the hard disk is accessed and it remains idle, i.e., the *pure* and *stable* conditions instantiated for the disk are fulfilled. Using Lemma 3 we can now prove correctness of part 1, by only analyzing the global computations without external device steps. Note, that it suffices to validate stability and purity only for the empty sequence.

The hard disk remains stable in buffer write state *bwr*, and purity still holds for the first inner loop. Hence, again by Lemma 3, it suffices to establish the invariant only for the empty sequence.

Things get more involved in the second inner loop, due to absence of stability: at an arbitrary time the hard disk may transfer the buffer content to the persistent sector memory. Hence, termination of the polling loop depends on the external environment, which finally indicates the operation to be completed. A full-blown interleaved analysis is still not necessary. Applying Theorem 1 device steps other than the hard disk can be ignored, and we establish the proof only over sequences reduced to disk steps. However, we first have to discharge the separability condition for the trigger restriction imposed by the disk on the environment. This follows from a simple application of Lemma 1.

Summarizing, except for the polling loop in part 4, correctness could be shown completely sequentially.

The driver presented here is used in Verisoft kernel code to perform page swap-out [22]. This is achieved by embedding it into a C function declared as `void write_to_disk(int a, int b)`. Two more instructions are needed to load the parameters from the program stack. These instructions do not access devices.

With the abstraction technique from Sect. 3, we formally established correctness of the driver call against an atomic specification. Using it the correctness of client code can be shown in Hoare logic.

6 Future Work

There are several directions for future work. The disk driver presented in Sect. 5 is used in Verisoft microkernel code for page swap-out [22]. The driver for page swap-in remains to be verified. Also, the driver given is only a polling one. For the file system implementation in Verisoft's simple operating system interrupts are also used. This code is being verified.

For the verification of code for device other than hard disk, it might be interesting to refine the concept of stability, which was introduced in Sect. 3. Typically, a communicating device (e.g., network interface card) is never stable on the complete configuration because it always asynchronously transfers data. However, by defining stability only for parts of the state, it can still usually be preserved. For example, communication is often channeled through buffers for transmission and reception with processor and environment accessing these buffers at different ends. Concurrency can be reduced in such a scenario.

7 Conclusion

We have presented the formal functional correctness proof of an assembly disk driver against a formal architecture model integrating devices, which is concurrent. The proof could be decomposed into two parts. First, we have proven the driver correct in a model with just the hard disk present. By abstracting from the other devices, the set of model runs has been reduced significantly. Second, we have generalized this result to computations in the full model by proving a general reordering theorem. This theorem is applicable if devices do not interfere with each other and a device is controlled exclusively by a single driver. The same reordering can also be applied to the high-level language model with devices, allowing to separate high-level computational from device steps.

Not classical problems such as finding correct invariants turned out to be hard during the verification process. Most notably, an appropriate program logic for assembly and better support for arithmetics in the prover were sorely missed. With the help of reordering, interleaved reasoning was only required for two lines of code, amounting to one third of the overall verification effort.

Combining our result with hardware and compiler correctness [20,23], allows to transfer properties of a high-level program calling our driver down to the gate-level implementation of the complete system.

References

1. Hillebrand, M.A., Paul, W.: On the architecture of system verification environments. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 153–168. Springer, Heidelberg (2008)
2. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
3. American National Standards Institute: ANSI NCITS 340-2000: AT Attachment-5 with Packet Interface (2000)

4. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD 2005, pp. 309–316. IEEE Computer Society, Los Alamitos (2005)
5. Alkassar, E., et al.: Formal device and programming model for a serial interface. In: Beckert, B. (ed.) VERIFY 2007, pp. 4–20 (2007)
6. Berry, G., Kishinevsky, M., Singh, S.: System level design and verification using a synchronous language. In: ICCAD, pp. 433–440. IEEE Computer Society / ACM, Los Alamitos (2003)
7. Rashinkar, P., Paterson, P., Singh, L.: System-on-a-Chip Verification: Methodology and Techniques. Kluwer Academic Publishers, Norwell (2001)
8. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
9. Microsoft Corporation: SDV: Static driver verifier (2004), http://www.microsoft.com/whdc/devtools/tools/sdv.mspx
10. Hallgren, T., et al.: A principled approach to operating system construction in Haskell. In: Danvy, O., Pierce, B.C. (eds.) ICFP. ACM, New York (2005)
11. Heiser, G., et al.: Towards trustworthy computing systems: Taking microkernels to the next level. SIGOPS Oper. Syst. Rev. 41(4), 3–11 (2007)
12. The FLINT Project, http://flint.cs.yale.edu/flint/
13. Holzmann, G.J.: New challenges in model checking. In: Symposium on 25 years of Model Checking, Seattle, USA, LNCS, vol. 4925. Springer, Heidelberg (August 2006)
14. Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/Eves: An experiment in the verified software repository. In: ICECCS, pp. 3–14. IEEE Computer Society, Los Alamitos (2007)
15. Butterfield, A., Woodcock, J.: Formalising Flash memory: First steps. In: ICECCS, pp. 251–260. IEEE Computer Society, Los Alamitos (2007)
16. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
17. Cohen, E., Lamport, L.: Reduction in TLA. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 317–331. Springer, Heidelberg (1998)
18. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
19. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. Form. Methods Syst. Des. 28(3), 263–289 (2006)
20. Tverdyshev, S., Shadrin, A.: Formal verification of gate-level computer systems. In: Rozier, K.Y. (ed.) LFM 2008. NASA STI, NASA, pp. 56–58 (2008)
21. Yu, D., Shao, Z.: Verification of safety properties for concurrent assembly code. In: ICFP 2004 (September 2004)
22. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 109–123. Springer, Heidelberg (2008)
23. Leinenbach, D., Petrova, E.: Pervasive compiler verification – From verified programs to verified systems. In: SSV 2008. ENTCS, vol. 217C, pp. 23–40. Elsevier Science B.V., Amsterdam (2008)

Verified Process-Context Switch for C-Programmed Kernels

Artem Starostin* and Alexandra Tsyban**

Computer Science Dept., Saarland University, Germany
{starostin,azul}@wjpserver.cs.uni-sb.de

Abstract. A context switch — an act of saving and restoring the state of a CPU such that multiple processes can share a single CPU resource — is an essential feature of multitasking operating systems. Commonly computationally intensive and necessarily accessing hardware registers, context-switch procedures are implemented as inline assembly portions in C-programmed operating-system kernels. Feasible verification of operating systems is usually attempted in some kind of C semantics. However, seamless verification of kernels requires reasoning about context-switch routines in semantics of assembly language. At the end of the day, both semantics meet together in an overall correctness theorem of operating system. The task of formal integration of correctness results achieved on different semantical layers is challenging but inevitable for systems verification.

The paper describes a formal approach to pervasive reasoning about interleaved computations of user processes and a C-programmed kernel. The interleaving is achieved by context-switch procedures implemented in inline assembly. We report on the correctness proof of the context-switch procedures and elaborate on our experience in formal integration of this result into the correctness proof of CVM, a verified framework for microkernel programmers.

1 Introduction

Pervasive systems verification [2] — an act of proving correct an entire computer system from gates to software — is a grand challenge [16] undertaken by the Verisoft project [21]. In our experience, the complexity of the problem turns out to be not in verification of individual components comprised by a system, but rather in *formal* integration of different correctness results achieved separately.

One representative example where different formal theories meet together is a correctness proof of process-context switch. Context-switch procedures save and restore the state (content of visible registers) of a CPU such that multiple processes can share a single CPU resource. This puts them at the heart

* Work was supported by the International Max Planck Research School for Computer Science (IMPRS-CS).

** Work was supported by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 240–254, 2008.

of multitasking operating systems. Convincing arguments about correctness of context switch tie together at least the following formal theories: (i) a model of processor on which user processes and an operating system run, (ii) a virtual memory model used to implement separate address spaces of user processes, (iii) some sort of C semantics used to model computations of operating system, and (iv) inline assembly semantics used to reason about context-switch routines as the latter access hardware registers not visible via C variables. These formal theories have been developed in the scope of Verisoft [4,5,15,20], however putting them together in a correctness theorem of context switch requires additional investigations. The complications appear when one needs to switch from a computation of model x to a computation of model y. In order to reason about y, it turns out that additional invariants about it have to be preserved under the computation of x. Determining such invariants and proving them is a tedious scientific task.

The paper describes a pervasive correctness proof of process-context switch. Our context-switch procedures have C signatures, their functionality is implemented as inline assembly portions. This makes possible to plug them into C implementations of operating-system kernels on the source code level. In order to guarantee correctness of context-switch procedures as parts of C-programmed kernels it is not sufficient only to show their functional correctness. In order to proceed with kernel computations additional invariants about the kernel have to hold during computations of user processes. We introduce a concept of *weak kernels* — together with a correctness relation towards a hardware model — a formalization of suspended kernels during user executions. Our correctness theorems of context-switch procedures formally describe how an active kernel is constructed out of weak one and vice versa. This result makes possible to produce a pervasive seamless proof of C-programmed microkernels, like those that are considered in Verisoft. The paper gives details on integration of correct context-switch procedures into CVM, a framework for building verified kernels [13]. All material presented in the paper is supported by formal theories[1] in Isabelle/HOL [19], a verification environment common to Verisoft.

We believe the contributions of the paper fit into the program of the Verified Software Initiative [10], a state-of-the-art plan for research communities with the goal of making verification a core technology for developing reliable software. The authors personally see contributions to two fields: (i) theory unification, and (ii) experience reports on verification of realistic software systems. The former is confirmed as follows. We have not only exploited existing formal theories, rather evolved and adjusted them in a way they *formally* fit together. We have found and formalized additional invariants that made the integration possible. We have produced a formal proof for a critical part of realistic operating-system microkernel running on a verified processor — this argues for the second contribution.

Related Work. Much and, to the best of our knowledge, pioneering work on formal verification of process-context switch has been done by the FLINT group

[1] Theory files are available at `http://www.verisoft.de/VerisoftRepository.html`.

at Yale University. [18] describes how XCAP — a verification framework implemented in the Coq proof assistant — is applied to certify an x86-assembly implementation of machine-context management. [7] shows the FLINT's approach to integration of context switches with thread implementation. The correctness criteria considered are formulated, in contrast to the current paper, only at the assembly-language level: no results on integration of object code correctness into a high-level programming language are reported. Conversely, a large number of research groups working on OS verification consider only code implemented in high-level programming languages. The Singularity project builds an OS using Sing#, a type-safe programming language [11]. The NICTA group considers C implementation generated from an OS prototype designed in Haskell [9]. Both rely on unsafe portions of assembly code for context-switch.

Outline. We start in Sect. 2 by defining computational models of CPU, user processes, and a kernel. In Sect. 3 we describe the interaction between the models and state our verification objective. We continue in Sect. 4 by outlining our verification approach and presenting the correctness theorems of context switch. Finally, we conclude in Sect. 5.

Notation. Let bv denote the set of all bit vectors of length 32. We denote a list with elements of type t as t^*. An empty list is denoted by []. A list x is accessed by $x[i]$. A sublist of a list x is $x[i..j]$. Concatenation of lists or list elements x and y is $x \circ y$. For a record x with components a, b, and c we agree on $x = x.(a, b, c) = (x.a, x.b, x.c)$. Record parts are accessed in the same fashion, e.g., $x.(a, b) = (x.a, x.b)$. For a natural number d we denote by $m_d(a)$ the content of d consecutive memory cells starting at address a.

2 Computational Models

We consider a scenario where a number of user processes interact with a kernel on a shared CPU resource. We use a *physical machine* [4] to model the CPU and *virtual machines* [5] to model user processes. We use the *C0 language* [15], a slightly restricted dialect of C developed in Verisoft, to implement the kernel. Note, that kernel implementation necessarily contains inline assembly portions as the kernel accesses hardware registers not visible to C0 variable. The C0 small-step semantics is used to model kernel computations. Next, we give details on each concept.

2.1 Physical Machine

The physical machine is a sequential abstraction of the VAMP hardware [4] — a formally verified processor with memory management units, out-of-order execution, and memory-mapped devices — as seen by a system software programmer. VAMP implements the DLX instruction set [17]. The model state pm is the record with the following components: (i) the normal $pm.pc :: bv$ and the delayed $pm.dpc :: bv$ program counters used to implement the delayed branch mechanism, (ii) the general purpose $pm.gpr :: bv^*$ and the special purpose $pm.spr :: bv^*$

Table 1. Special purpose registers

Alias	Name	Description
sr	status reg.	stores an interrupt mask
esr	exceptional status reg.	stores the last interrupt mask before an interrupt
eca	exceptional cause reg.	stores masked interrupt cause
epc	exceptional *pc*	stores the last *pc* value before an interrupt
edpc	exceptional *dpc*	stores the last *dpc* value before an interrupt
edata	exceptional data reg.	stores data specific for a certain interrupt
pto	page-table origin	origin in the page table for address translation
ptl	page-table length	number of allocated physical memory pages
mode	mode reg.	value 0 is system, any other value is user

register files, and (iii) the byte-addressable physical memory $pm.m :: bv \mapsto bv$. The general purpose register file contains 32 registers: $pm.gpr[0]$ always contains zeros, and the last three registers are reserved for the global, the stack, and the heap pointers of a program. We abbreviate them as $pm.gp$, $pm.sp$, and $pm.hp$. The special purpose register file contains 9 registers described in Table 1. For each alias x we abbreviate access to the special register $pm.spr[x]$ as $pm.x$.

The computation of the model is defined by the transition function δ_{pm} which maps the current processor configuration pm and a bit vector of external inputs eev to the next configuration $pm' = \delta_{pm}(pm, eev)$. Due to the lack of space we do not present a complete definition of the step function, but rather an idea behind it. For formal details cf. [3]. Executions are possible in two modes: user and system. The processor switches to the system mode, denoted by $sys(pm)$, when an interrupt, either internal or external, arrives and the JISR (jump to interrupt service routine) signal, denoted by $jisr(pm, eev)$, is activated. The mode switch in the opposite direction happens when the RFE (return from exception) instruction, denoted by $rfe(pm)$, is executed. In user mode a memory address a is virtual and is subject to address translation. The translation either redirects to the computed physical memory address or generates a *page fault* interrupt which signals that the desired page is not in the physical memory. The decision is made by examining the *valid* bit $v(pm, a)$ stored in *page tables* maintained by memory management units of the physical machine. When on, it signals that the page storing the virtual address a resides in the main memory, otherwise, it is on a hard disk.

Interrupts. Internal interrupts comprise an illegal instruction, a misaligned access, page faults, overflows, and the *trap*. The latter is used to implement system calls to kernel libraries. External interrupts are signals from devices. VAMP supports maskable interrupts. For a maskable interrupt i, the bit $pm.sr[i]$ stores its mask. The *cause* register $ca(pm, eev)$ is the bit vector which stores raised interrupt signals. The *masked cause* register $mca(pm, eev)$ is defined as a bitwise conjunction $ca(pm, eev)[i] \wedge pm.sr[i]$ over all maskable interrupts i. If at least one bit of $mca(pm, eev)$ is on the $jisr(pm, eev)$ is activated.

Semantics of JISR and RFE. On $jisr(pm, eev)$, the program counters $pm.(dpc, pc)$ are set to the kernel start address $(kstart, kstart + 4)$. The exceptional versions of registers $pm.(edpc, epc, esr)$ are assigned their normal versions. Register $pm.eca$ stores the masked interrupt cause $mca(pm, eev)$. Finally, $pm.edata$ stores data needed to process a particular kind of an interrupt, e.g., an address of a page missing in the main memory for page-fault interrupts, or a system call code for the trap interrupt. The mode is switched to system. On $rfe(pm)$ the normal versions of registers $pm.(dpc, pc, sr)$ are assigned their exceptional versions. The mode is switched to user.

2.2 Virtual Machines

A virtual machine is a hardware abstraction with an illusion of address space exceeding the physical memory. The state of a virtual machine vm is a record comprising the same components as the physical machine. There are different rules for accessing special purpose registers: only reading is allowed. As for the rest, the semantics of virtual machines simply implements the DLX instruction set architecture. There is no address translation, hence page faults are invisible. There is no interaction with devices. Transitions are modeled by the function δ_{vm} which takes the current virtual machine configuration vm and yields an updated one $vm' = \delta_{vm}(vm)$.

2.3 C0 Small-Step Semantics

The C0 concrete syntax and visibility rules for variables are very similar to standard C. Operational semantics, though, is similar to Pascal. The main language restrictions compared to C comprise absence of pointer arithmetic, side effects, and pointers to functions and local variables. A list of assembly instruction il can be embedded in C0 by a special statement $asm(il)$.

A C0 program π is a record with the following components: (i) a global symbol table $\pi.gst$ which is a list of variable names together with their types, (ii) a type name environment $\pi.te$ which maps type names to types, and (iii) a function table $\pi.ft$ which maps function names to functions. A function is represented by symbol tables for parameters and local variables, the function's return type, and the function's body.

Configurations c of the C0 small-step semantics are records with two components: the memory configuration $c.mem$ and the program rest $c.pr$. The program rest stores those statements which still have to be executed. The memory configuration is a record comprising the following components: (i) a global memory frame $c.mem.gm$, (ii) a stack of local memory frames $c.mem.lm$, and (iii) a memory frame for heap variables $c.mem.hm$. A memory frame m consists of a symbol table $m.st$ which lists the variables of the frame and of a content function $m.ct$ which maps addresses to memory cells. A single memory cell can store values of elementary types. Values of aggregate types are stored flattened as a consecutive sequence of memory cells. Executions of the C0 small-step semantics are modeled by the function δ_{C0} which yields for a given C0 configuration c and a

program π either a special error state or an updated configuration $c' = \delta_{\mathrm{C0}}(c, \pi)$. A computation ends in the error state if, for instance, expression evaluation detects an error in the program, e.g., null-pointer dereference, or out-of-boundary array access. Details are available through [14].

3 Verification Objective

CVM (communicating virtual machines) [8] is a computational model for concurrent user processes interacting with a generic microkernel and devices. CVM is implemented in C0 with inline assembly as a framework featuring virtual memory support, memory management, and low-level inter-process and devices communications. The framework can be linked on the source code level with an abstract kernel, an interface to users, in order to obtain a concrete kernel, a program that can run on a target machine, like the kernel we discuss in the paper. CVM has been verified to a large extent within Verisoft [13].

Let a C0 with inline assembly program π_k be a concrete kernel implementation. Let k be a C0 small-step configuration corresponding to π_k. The kernel interacts with N user processes $up[1..N]$. Each process $up[i]$ is modeled by a virtual machine. When a user execution is interrupted the hardware generates the JISR signal which triggers the kernel start. Kernel executions begin with the SAVE procedure which stores the content of the hardware registers to kernel variables. After that, depending on the interrupt, kernel performs one or several jobs: handles interrupts, invokes system calls, schedules the next process. Kernel execution ends with the call of the RESTORE procedure which copies the context of the scheduled user process to hardware registers and executes the RFE instruction.

The CVM verification objective is to justify correctness of (pseudo-)parallel executions of user processes and the kernel on the underlying hardware. This is expressed as a simulation theorem between the physical machine and virtual machines interleaving with the kernel: for all steps of an interleaved execution of the kernel and user processes there exists a number of hardware steps after which an appropriate simulation relation holds. The simulation relation is split into two parts: each relates a hardware configuration either to a kernel configuration, or to configurations of user processes. One of the most interesting proof cases is switching between executions of different user process by means of a kernel invocation. The task is tricky since correctness theorems of SAVE and RESTORE (i) tie together different semantics used to model user processes and the kernel, (ii) formalize a connection between correctness relations for users and the kernel, and (iii) involve inline assembly semantics to prove the functional correctness.

Correctness criteria of user processes must reflect two main features realized by CVM: memory virtualization and multitasking. Correct memory virtualization ensures that each user process has a notion of its own isolated virtual memory exceeding the physical memory. CVM implements memory virtualization by means of demand paging: [1] reports on formal verification of the used paging mechanism. Multitasking is a method by which multiple processes share a

single computational resource like CPU. Correct multitasking ensures reassigning a CPU from one process to another. This paper focuses on correctness of multitasking implementation in CVM.

The rest of this section is organized as follows. We give some details on the kernel implementation, discuss the notion of contexts, and describe our implementation of context-switch procedures. Next, we state the correctness relations for the kernel and user processes and describe at which parts of an interleaved run which relations hold. Finally, we discuss how to combine correctness criteria by introducing invariants over a suspended kernel.

3.1 Kernel

We consider a non-preemptive kernel. On a user interrupt the kernel needs to know which process has been preempted. When RESTORE is invoked the kernel needs to know which process has to be resumed. In order to deal with it, the kernel has the variable *cup* of unsigned integer type storing the identifier of the last executed process. The kernel scheduler assigns the new value to *cup*, thus the process $up[cup]$ will be resumed.

We allow dynamic memory allocation for the kernel. The kernel contains the variable *kheap* of integer type which stores the amount of memory allocated by the kernel on the heap. Base addresses of kernel global and stack memories are defined by the constants *kglobal* and *kstack*, respectively.

3.2 Contexts and Context-Switch Procedures

We split registers of user processes into two groups: those whose values affect user executions and those that are used only to implement the interrupt mechanism.

Registers Relevant for User Executions. This group comprises delayed and normal program counters, 31 general purpose registers excluding $gpr[0]$ — according to the DLX instruction set architecture it always contains zeros —, the status register, page-table origin, and page-table length registers. We call these registers the *context* of a process. For a virtual machine *vm* it is formally retrieved by the function $ctx(vm) = vm.pc \circ vm.dpc \circ vm.gpr[1..31] \circ vm.sr \circ vm.pto \circ vm.ptl$. We store contexts of processes in a special data structure, called the *process control blocks*. The kernel symbol table $\pi_k.gst$ includes the variable *pcb*, an integer array with $36 \cdot N$ items capable to store all contexts of user processes.

Registers Needed to Process Interrupts. In order to react appropriately to an interrupt which occurs during a user execution the kernel has to know the interrupt cause and exceptional data, e.g., if an interrupt happens due to the trap instruction the kernel has to know its parameter in order to execute the right system call. Because of that we declare two variables in π_k: (i) the variable *eca* of integer type used to store the current exceptional cause register, and (ii) the variable *edata* of integer type storing data for interrupt handling.

The context-switch procedures are C0 functions with inline assembly bodies. The SAVE procedure starts by computing the memory address $a{=}4{\cdot}\&(pcb[cup{\cdot}36])$.

Starting at address a we write to the memory $pm.m$ consecutively the contents of the following registers: (i) $pm.epc$ and $pm.edpc$ as they are equal to $pm.pc$ and $pm.dpc$, and (ii) $pm.gpr[1..31]$. Note, that we do not save registers $pm.(sr, pto, ptl)$ because they could not be modified by a user. Next we save registers $pm.eca$ and $pm.edata$ to corresponding kernel variables and assign $pm.gp$, $pm.sp$, and $pm.hp$ the values of $kglobal$, $kstack$, and $kheap$. The last statement of the SAVE routine is a C0 call to the kernel dispatcher.

The RESTORE function is quite symmetrical to SAVE. In the C0 portion of the function we compute an offset in pcb corresponding to the next scheduler process identifier cup. By means of inline assembly we remember the value of the kernel heap pointer in variable $kheap$ and write the context of the next scheduled process from pcb into hardware registers $pm.(epc, edpc, gpr[1..31], esr, pto, ptl)$. Note, that $pm.sr$, $pm.pto$ and $pm.ptl$ registers are updated as the kernel can modify their values, e.g., when it allocates additional memory for some user process. The last instruction of RESTORE is RFE.

Assembly bodies of the context-switch procedures contain all in all about 100 instructions.

3.3 Kernel Correctness Relation: C0 Consistency

Our correctness relation for the kernel maps a C0 small-step semantics state to a state of the physical machine. A C0 configuration c corresponding to a program π is related to an underlying physical machine pm by the compiler simulation relation $consis(alloc)(\pi, c, pm)$ parameterized over an allocation function $alloc$ which maps C0 variables to the physical memory cells. The relation describes consistency of (i) code, (ii) control, and (iii) data. The code consistency $consis_{\text{code}}$ requires that the compiled code lies at the correct address in the memory $pm.m$. The control consistency $consis_{\text{c}}$ states that the delayed program counter $pm.dpc$ points to the start of the translated code of the first statement of $c.pr$ and that return addresses of all stack frames are correct. Essentially, the data consistency $consis_{\text{d}}$ is a conjunction of the correctness relation for: (i) allocation, $consis_{\text{alloc}}$: the allocation function $alloc$ conforms with the allocated base address of global and local variables, and the reachable portions of the heaps in $c.hm$ and $pm.m$ are isomorphic, (ii) values and pointers, $consis_{\text{v}}$ and $consis_{\text{p}}$: the respective variables and pointers of c and pm have the same values, (iii) registers, $consis_{\text{r}}$: the heap pointer $pm.hp$ points to the first free address of $c.mem.hm$, and the global resp. the stack pointers $pm.gp$ resp. $pm.sp$ refer to the beginning of the global resp. top local memory frames of c. A scheme of the C0 consistency relation is depicted in Fig. 1. For complete formal definitions cf. [14].

3.4 Correctness of User Processes: Context-Encoding Relation

The correctness of user processes is stated by the context-encoding relation $\mathcal{C}$ which maps the vector of user processes up to the physical machine configuration

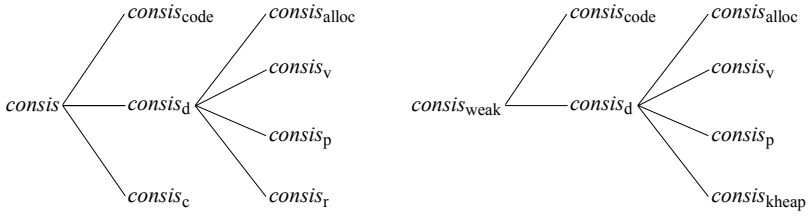


Fig. 1. C0 consistency (left) and C0 weak consistency (right) relations

pm. For a process pid and a register r which belong to the context, let $ad^r_{pid}(pm)$ be the physical memory address in the machine pm where the register r of the process pid is stored in the pcb array. The content of this register is then $r_{pid} = pm.m_4(ad^r_{pid}(pm))$. If r refers to a general purpose register i of a process pid we write $gpr[i]_{pid}$. Also we define $gpr[i..j]_{pid} = gpr[i]_{pid} \circ \ldots \circ gpr[j]_{pid}$. The address and the content of the cup variable are $ad^{cup}(pm)$ and $cup(pm) = pm.m_4[ad^{cup}(pm)]$, respectively. For an inactive process pid we define its stored context as $stored\text{-}ctx(pm, pid) = pc_{pid}(pm) \circ dpc_{pid}(pm) \circ gpr[1..31]_{pid}(pm) \circ sr_{pid}(pm) \circ pto_{pid}(pm) \circ ptl_{pid}(pm)$. In case $sys(pm)$ the kernel is running and all user process are suspended. Otherwise a user with identifier $pid = cup(pm)$ is running and its context occupies physical registers of the hardware. We define the context of a physical machine pm as $active\text{-}ctx(pm) = pm.pc \circ pm.dpc \circ pm.gpr[1..31] \circ pm.sr \circ pm.pto \circ pm.ptl$. Altogether, we formalize both stored and active process contexts as:

$$proc\text{-}ctx(pm, pid) = \begin{cases} stored\text{-}ctx(pm, pid) & \text{if } sys(pm) \vee cup(pm) \neq pid \\ active\text{-}ctx(pm) & \text{otherwise} \end{cases}.$$

The context-encoding relation states that all user processes are simultaneously encoded by the hardware:

$$\mathcal{C}(up, pm) = \forall pid : proc\text{-}ctx(pm, pid) = ctx(up[pid]).$$

3.5 Combining Correctness Relations

The correctness of user processes, expressed by the context-encoding relation, has to hold at every step of user executions. A kernel run might contain a system call which affects users, e.g., writing to a user register. The abstract semantics, i.e., specification, of system calls is defined over the vector of user processes up. The concrete semantics, i.e., implementation, affects though the underlying physical machine. Abstract and concrete semantics must agree: universally this is expressed as the the context-encoding relation must be preserved under kernel runs.

The correctness of the kernel, expressed by the C0 consistency relation has to hold after an execution of every statement of the kernel. However, a C0 small-step semantics configuration encoding the kernel and consistent to the underlying

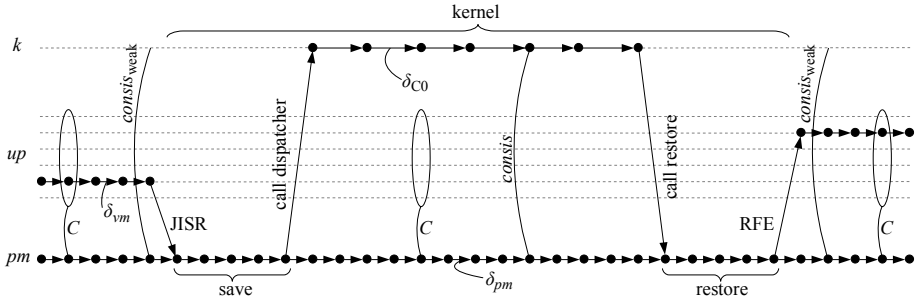


Fig. 2. Overall correctness: context-encoding relation holds throughout users and kernel executions, C0 consistency holds during kernel runs, and C0 weak consistency holds during users computations

hardware cannot be simply constructed from a physical machine configuration after a user execution. There are two reasons for that: (i) there is no information about C0 types in the memory model of the physical machine, and (ii) there is no one-to-one mapping between the heap of the physical machine and the heap memory frame of the C0 configuration. In order to be able to resume a C0 kernel computation after a user execution users have to respect the constraint that they do not affect kernel code and kernel data, including a mapping between heaps. We call this constraint the *C0 weak consistency* and define formally below. Fig. 2 depicts overall correctness.

3.6 Correctness of Suspended Kernels: C0 Weak Consistency

The parts of the C0 configuration k encoding the kernel that is possible to preserve unchanged during user computation are the global memory frame $k.mem.gm$ and the heap memory frame $k.mem.hm$. These two components define a C0 small-step semantics configuration which we call a *weak kernel*. Formally, we construct a weak kernel k_{weak} by means of the function $\mathit{weak\text{-}kernel}(gm, hm) = k_{\text{weak}}$, s.t. $k_{\text{weak}}.mem = (gm, [], hm)$ and $k_{\text{weak}}.pr = \mathit{skip}$.

A weak kernel k_{weak} is related to a physical machine configuration pm w.r.t. the kernel program π_k and an allocation function *alloc* by the C0 weak consistency relation $\mathit{consis}_{\text{weak}}(\mathit{alloc})(\pi_k, k_{\text{weak}}, pm)$. The relation comprises: (i) the code consistency $\mathit{consis}_{\text{code}}$, which guarantees that users do not modify the kernel code, (ii) the value consistency $\mathit{consis}_{\text{v}}$ and the pointer consistency $\mathit{consis}_{\text{p}}$, which guarantees that users do not modify the kernel data, and (iii) the allocation consistency $\mathit{consis}_{\text{alloc}}$ and the kernel heap consistency $\mathit{consis}_{\text{kheap}}$, which defines the mapping between heaps of the physical machine and the C0 configuration. The kernel heap consistency might be seen as a part of the register consistency: exploiting the fact that the variable *kheap* stores the value of the kernel heap pointer, $\mathit{consis}_{\text{kheap}}$ states that the value in *kheap* points to the first free address of $k_{\text{weak}}.mem.hm$. Fig. 1 compares normal and weak C0 consistency relations.

4 Correctness of Process-Context Switch

In this section we state the correctness theorems of process-context switch. The general idea behind their proofs is to conclude functional correctness, i.e., an input-output relation over configurations of the physical machine running the kernel, by means of inline assembly semantics and to show that the functional correctness result implies all necessary relations. Next, we present our approach to verification of inline assembly portions in C0 code. For more detailed description cf. [20].

4.1 Formal Inline Assembly Semantics

In brief, our approach to deal with verification of assembly statements is as follows. On an assembly statement the C-execution is no longer possible, thus we switch to the consistent underlying physical machine and continue the computations directly there. When all assembly instructions have been executed we switch back to the C0 level. In order to regain C0 consistency we have to reflect effects of assembly instructions on the C0 small-step configuration. Examining the allocation function *alloc* we determine which variables of the C0 configuration have changed. We retrieve their values from the physical machine and write back to the C0 memory configuration.

Let c be the C0 configuration with $c.pr = asm(il); r$, and let pm be the physical machine consistent to c, i.e., $consis(alloc)(\pi, c, pm)$. From the control consistency we have that the delayed program of pm point to the address of the assembly statement: $pm.dpc = ad(asm(il))$, where $ad(s)$ yields for a statement s its address in the memory of pm. This makes it possible to start reasoning about the correctness of the assembly code il directly in the semantics of the physical machine. Let pm' be the physical machines configuration after executing il. In order to formally specify the effect of an execution of $asm(il)$ on the C0 configuration c we define the function $upd(c, pm, pm') = c'$ which analyzes the difference between pm and pm' and projects it to the C0 level updating the configuration c to c'. A number of restrictions are imposed on the changes in the physical machine, which guarantee that the C0 configuration is not destroyed by the assembly portion il, namely: (i) the program pointers after the execution of il point to the end of il: $pm'.dpc = pm.dpc + 4 \cdot |il|$, (ii) the global, stack, and heap pointers are unchanged: $pm'.(gp, sp, hp) = pm.(gp, sp, hp)$, (iii) the memory region where the compiled code is stored stays the same, i.e., we forbid self-modifying code, (iv) the memory occupied by the local memory frames remains the same except for top local memory frame, and (v) pointers change is forbidden except setting them to *null*. The inline assembly semantics forces us to prove that assembly portions meet these restrictions.

It is easy to notice, that assembly implementations of context-switch procedures do not meet several restrictions. Both, Save and Restore procedures rewrite global, stack, and heap pointers, and therefore violate restriction ii. The Restore routine has RFE as the last instruction — its semantics rewrites program counters, which violates the first restriction. Fortunately, Save is the first

function of the kernel and RESTORE is the last. Moreover, there is no C0 statements in SAVE before, and in RESTORE after their assembly portions. Thus, we do not construct C0 configurations before the assembly portion of SAVE, and after the assembly portion of RESTORE. As long as there is no complete C0 configurations the stated restrictions are irrelevant. Certainly, we need to have a C0 configurations after the assembly portion of SAVE in order to proceed with a kernel C0 computation. For this we project the semantics of assembly parts on the memory of a weak kernel and construct the desired C0 configuration from it. The details are given in the proof sketch of the correctness theorem for SAVE.

4.2 Correctness Theorems of Process-Context Switch

We start with a simpler case of context restore: the next theorem justifies a switch from a kernel to a user executions.

Theorem 1 (Correctness of Process-Context Restore). Let k be a C0 small-step configuration of the kernel program π_k, up a vector of user processes configurations, and pm a configuration of the underlying physical machine. Assume that (i) the kernel is invoking the RESTORE procedure: $k.pr = call(\text{RESTORE})$, (ii) the kernel is consistent to the physical machine: $consis(alloc)(\pi_k, k, pm)$, and (iii) the context-encoding relation holds: $\mathcal{C}(up, pm)$, then there exists a number of steps T of the physical machine, s.t. $pm' = \delta^T_{pm}(pm)$ after which (i) a user execution starts: $\neg sys(pm')$, (ii) the context-encoding relation is preserved: $\mathcal{C}(up, pm')$, and (iii) there exists a weak kernel configuration k_{weak}, s.t. it is weak-consistent to the physical machine: $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm')$.

Proof Sketch. Executing the assembly body of RESTORE in the semantics of the physical machine we obtain the number of steps T. By showing the functional correctness of RESTORE we obtain all necessary arguments to conclude $\neg sys(pm')$ and $\mathcal{C}(up, pm')$. We project the semantics of the assembly portion to global and heap memories $k.mem.gm$ and $k.mem.hm$ obtaining the new values gm' and hm'. We construct the weak kernel $k_{\text{weak}} = weak\text{-}kernel(gm', hm')$. From the initial C0 consistency relation $consis(alloc)(\pi_k, k, pm)$ it is fairly easy to conclude the desired weak consistency $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm')$.

The theorem for context save describes the computation which starts directly after the last user step: the JISR semantics is applied on the physical machine. However, since the mode register of the physical machine is set to system, the context-encoding relation does not hold at this point. In order to regain it we have to artificially undo the JISR effect. For this we define the function $\delta^{\text{-}jisr}_{pm}(pm)$ which returns the updated configuration pm' of the physical machine, s.t. $pm'.(pc, dpc, sr, mode) = pm.(epc, edpc, esr, 1)$.

Theorem 2 (Correctness of Process-Context Save). Let k_{weak} be a weak kernel configuration corresponding to the kernel program π_k, up a vector of

user processes configurations, and pm a configuration of the underlying physical machine. Assume that (i) the program counters point to the kernel start: $pm.(dpc, pc) = (kstart, kstart + 4)$, (ii) the suspended kernel is weak-consistent to the physical machine: $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm)$, and (iii) the context-encoding relation holds with the physical machine configuration on which the JISR effect is undone: $\mathcal{C}(up, \delta_{pm}^{\text{-}jisr}(pm))$, then there exists a number of steps T of the physical machine, s.t. $pm' = \delta_{pm}^{T}(pm)$ after which (i) the context-encoding relation is preserved: $\mathcal{C}(up, pm')$, and (ii) there exists a kernel C0 small-step configuration k, s.t. the consistency relation $consis(alloc)(\pi_k, k, pm)$ holds.

Proof Sketch. The code consistency conjunct of $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm)$ ensures that the translated code of SAVE resides in the memory of the physical machine at address $kstart$. By executing the assembly body of SAVE in semantics of the physical machine we get the number of steps T. From the functional correctness of the SAVE code we conclude the $\mathcal{C}$-relation. We project modifications over $k_{\text{weak}}.mem.gm$ and $k_{\text{weak}}.mem.hm$ done in assembly semantics to the C0 level and obtain the new values gm' and hm'. We specify a single local memory frame lm for the SAVE function and define a program rest pr containing a single statement — a call to the kernel dispatcher. The resulting C0 small-step configuration k is constructed as $k.mem = (gm', lm, hm')$ and $k.pr = pr$. We exploit the initial weak consistency $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm)$ to show the coincident conjuncts of $consis(alloc)(\pi_k, k, pm')$. The register consistency conjunct follows from the fact that we have loaded kernel values to $pm'.(gp, sp, hp)$. Execution of the assembly body on the physical-machine level sets the program counters to the address of the first C0 statement, thus we conclude the remaining control consistency.

4.3 Considering Additional Invariants

We have presented how CVM implements correct multitasking by means of verified process-context switch. Being a part of the kernel, context-switch procedures must also respect correctness of memory virtualization. Next, we describe additional invariants which we consider in order to guarantee that.

Memory Virtualization Relation. Memories of user processes are stored partly in the memory of the physical machine and partly on the hard disk. In case user process pid accesses address a with the corresponding page present in the physical memory, i.e., valid bit $v(pm, a)$ is on, the kernel provides the translated physical memory address $pma(a, pid)$. Otherwise, the page resides in the swap memory of the hard disk $hd.sm$ at the swap memory address $sma(a, pid)$. It is subject to move to the main memory by the page-fault handler. We reconstruct the virtual memory of a user process with an identifier pid by means of the function $proc\text{-}m(pm, hd, pid)$:

$$proc\text{-}m(pm, hd, pid) = \lambda a : \begin{cases} pm.m(pma(a, pid)) & \text{if } v(pm, a) \\ hd.sm(sma(a, pid)) & \text{otherwise} \end{cases}.$$

The memory virtualization relation, called the $\mathcal{B}$-relation is defined as follows:

$$\mathcal{B}(up, pm, hd) = \forall pid : proc\text{-}m(pm, hd, pid) = up[pid].m.$$

Proving $\mathcal{B}$-relation over Context Switch. The relation for memories might be affected by the assembly portions in SAVE and RESTORE. In order to show formally that this is not the case we prove that assembly code does not store in: (i) physical memory region occupied by virtual memories of user processes, (ii) page tables — this guarantees correctness of address translation, and (iii) hard disk ports — this argues that the data on the hard disk remains unchanged.

5 Summing Up

We have presented the first formal correctness proof of process-context switch for pervasively verified C-programmed kernels. We have stepped beyond existing approaches to reasoning about context switch, and inline assembly systems code in general, only in single semantics of underlying machine model because it is unacceptable for pervasive verification of operating systems. We have unified separate models — therefore, formal theories — for user processes, a kernel written in C with inline assembly, and hardware in correctness theorems for context switch. By that, a solid foundation for a trustworthy multitasking is lain. Our result is successfully integrated into formal theories of CVM, a verified framework for building kernels. Based on CVM — thus, exploiting the presented verified context-switch mechanism — two microkernels have been built, tested, and verified to a large extent in Verisoft: (i) VAMOS [6], an L4-inspired microkernel, and (ii) OLOS [12], an OSEKtime-like operating system, used in a distributed automotive real-time system establishing eCall functionality.

The formal proof of Theorem 1 conducted in Isabelle/HOL and its integration into the CVM model are about 5K steps long. The same task for SAVE, i.e., Thereom 2, is accomplished in 10K steps: the problem of constructing a kernel C0 state after a user execution required much effort, also correctness proof of physical memory modifications performed in SAVE is involved. The above numbers include 1.5K resp. 2K lines of interactive proofs for the functional correctness of RESTORE resp. SAVE assembly bodies. We believe that the latter numbers could come to naught by integrating automatic verification tools like assembly model checkers into Isabelle/HOL.

References

1. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 109–123. Springer, Heidelberg (2008)
2. Bevier, W.R., Hunt Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. Journal of Automated Reasoning 5(4), 411–428 (1989)
3. Beyer, S.: Putting It All Together: Formal Verification of the VAMP. PhD thesis, Saarland University, Computer Science Dept. (2005)

4. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Putting it all together: Formal verification of the VAMP. Intl Journal on Software Tools for Technology Transfer 8(4–5), 411–430 (2006)
5. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 301–316. Springer, Heidelberg (2005)
6. Daum, M., Dörenbächer, J., Wolff, B., Schmidt, M.: A verification approach for system-level concurrent programs. In: VSTTE 2008. LNCS, vol. 5295. Springer, Heidelberg (2008)
7. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 170–182. ACM, New York (2008)
8. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
9. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: Taking microkernels to the next level. Operating Systems Review 41(4), 3–11 (2007)
10. Hoare, T., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: A manifesto (2007), http://qpq.csl.sri.com/vsr/manifesto.pdf
11. Hunt, G.C., Larus, J.R.: Singularity: Rethinking the software stack. Operating Systems Review 41(2), 37–49 (2007)
12. der Rieden, T.I., Knapp, S.: An approach to the pervasive formal specification and verification of an automotive system. In: 10th Intl Workshop on Formal Methods for Industrial Critical Systems, pp. 115–124. IEEE, Los Alamitos (2005)
13. der Rieden, T.I., Tsyban, A.: CVM — a verified framework for microkernel programmers. In: 3rd Intl Workshop on Systems Software Verification. ENTCS, vol. 217, pp. 151–168. Elsevier Science B.V, Amsterdam (2008)
14. Leinenbach, D.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Computer Science Dept (2008)
15. Leinenbach, D., Petrova, E.: Pervasive compiler verification – from verified programs to verified systems. In: 3rd Intl Workshop on Systems Software Verification. ENTCS, vol. 217, pp. 23–40. Elsevier Science B. V, Amsterdam (2008)
16. Moore, S.J.: A grand challenge proposal for formal methods: A verified stack. In: Aichernig, B.K., Maibaum, T.S.E. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 161–172. Springer, Heidelberg (2003)
17. Müller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer, Heidelberg (2000)
18. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 189–206. Springer, Heidelberg (2007)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS. Springer, Heidelberg (2002)
20. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: 3rd Intl Workshop on Systems Software Verification. ENTCS, vol. 217, pp. 169–185. Elsevier Science B. V, Amsterdam (2008)
21. The Verisoft Consortium. The Verisoft Project (2003), http://www.verisoft.de

Where Is the Value in a Program Verifier?

Colin O'Halloran

Systems Assurance Group, Trusted Information Management, QinetiQ, UK
cmohalloran@qinetiq.com

Abstract. This paper addresses the assumption that software verification is valuable. Using experience from the assessment of safety critical software it makes observations of where the value of a Program Verifier might be and how it relates to the use of Formal Methods in requirements and design activities.

1 Do We Need Software Verification?

There is plenty of evidence that poor or immature development processes have led to poor software products. More often commercial pressures lead to trade-offs between the time (and cost) spent developing software and the dependability of the final software product. Unfortunately the converse assertion that good and mature software development processes lead to dependable products does not necessarily hold. For example the European Space Agency's well regarded processes for software development were adopted in the Ariane 5 programme, but it still failed spectacularly on its first launch due to a software fault [8].

The levels of reliability for Typhoon are so great that the provision of independent formal verification can be justified. Unfortunately the evidence that the formal proof provides is qualitative in nature, it does not empirically demonstrate the quantitative software reliability targets – typically of the order of 10^{-9} faults per operational hour. Of course neither can any other development method scientifically demonstrate such a software reliability target before the software has been operating without change for many years.

The aim of producing software with zero coding errors that a program verifier could achieve is not scientifically demonstrable before the software is deployed. Evidence also suggests that when systems fail it is because of poorly understood requirements or mistaken assumptions in the specification, or design. This suggests that the application of Formal Methods is most powerful in the areas of requirements, specification and design. The Program Verifier challenge would seem to not address a significant problem. The purpose of this paper is to examine the issues around the value of software verification.

2 The Typhoon Experience

The Systems Assurance Group at QinetiQ in Malvern has used mechanical proof techniques to verify three successive versions of Typhoon's flight control laws implemented in Ada. Approximately 37,000 lines of code were checked against three different Simulink[1] specifications scheduled over 3 processors. Each evolution of the

[1] Simulink is a graphical language for control system specification with simulation and code generation tool support.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 255–262, 2008.

control laws led to a change to about a third of the Ada source code each time. The details of the verification process are described in [7].

Just over 97% of the verification conditions were automatically proven. The remaining 3% were either manually discharged using a theorem prover, or could not be proven due to limitations in the tools at that time, or very occasionally were false due to mismatches between the specification and the code. Most of theses mismatches were due to the specification not being updated (the code in fact correctly implemented the requirements).

Two important points arise from this experience: the first is that these implementations had already been subjected to a mature software verification and validation process when only once did a coding error slip through2; the second is that the requirements for the flight control laws have been continually evolving in the light of flight tests and increased aircraft capability.

Further verification work, on more recent sub-systems, led to more than 80% of the verification conditions for Typhoon's autopilot and auto-throttle being automatically proven. The remaining verification conditions were subject to informal "hand proof"[3].

3 A Software Development Cost Model

Program code is the ultimate expression of requirements, specification and design. Although conventional software development can prove to be very effective, it has also proven to be expensive. The cost of conventional development and testing also increases considerably as the consequence of failure grows. Consider the following model of software development[4] for a system in Figure 1 below.

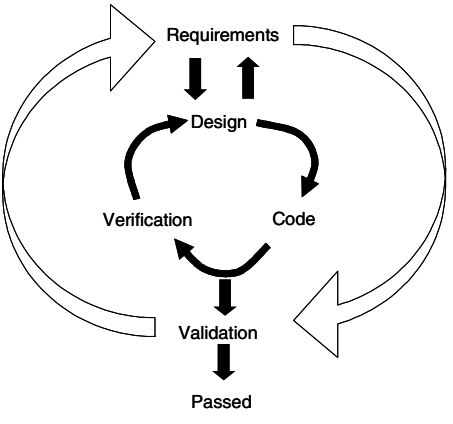


Fig. 1.

[2] The coding error was revealed by QinetiQ's verification method, but turned up first during rig testing that had started 6 months beforehand.

[3] The automated proof has been less successful because the specifications of these applications are not as mature as the specification for the flight control laws and hence contains more small errors. Thus an unexpected benefit is that the specification is validated.

[4] I am indebted to Tony Powell, of the University of York's department of management studies, for introducing me to this model.

The model starts with the requirements, which might be captured by a model that can be simulated. The simulations would help explore the requirements space and lead to changes in the model, a different articulation of the requirements, or actual changes to the requirement in the light of insights gained. Moving around the loop in the model of the development is relatively inexpensive. At some point, after N iterations of the outer loop, a development will enter the design phase leading to the inner loop in the model shown in Figure 1. Code is developed from the design, verified by some means (testing, Fagan inspections, or whatever) and then probably corrected.

After M iterations of the inner loop of the development, a judgement will be made that no more is needed and it will move back to the outer loop. At this point requirements testing will take place. Requirements testing at this stage in the development can be a very expensive affair possibly employing continual operation of test rigs and test flights (in the case of aircraft). An error[5] found at this stage will mean that a transition will be made from the outer loop back into the inner loop to correct the identified error. Unfortunately more inner loop iterations may be necessary before moving back to the outer loop to repeat requirements testing. After P iterations of the outer loop a judgement will be made that it has passed and the product can be deployed. The cost of a residual error not detected within the inner loop is greatly magnified by the requirements testing in the final outer loop and it leads to further iteration of the inner loop, possibly introducing more errors.

Two observations arise from this discussion. The first observation is that the elimination of implementation errors would mean that any errors detected in the final outer loop must be requirements errors. If a Program Verifier were used, then only one iteration within the inner loop would be required; further there would be a significant reduction in costs due to the elimination of residual implementation errors that otherwise would be detected within the outer loop of requirements testing. The second observation is that the elimination of requirements errors in general has a larger impact than the elimination of implementation errors.

What does this mean for the Program Verifier grand challenge? First the use of formal techniques supported by tools is important for the reason given above. However how is a requirement error determined? Ultimately it is a subjective judgement (supported by a rational argument) that is made because real requirements are rooted within a human organisation. Even the requirements for an advanced aircraft like Typhoon are set by socio-economic and political conditions. Technical requirements based on capabilities derived from an operational analysis still require human judgement about for example acceptable loss rates or cost-benefit trade-offs. This means that people with special expertise will be central in judging requirements errors. Formal techniques and tools have an important role to make them more effective.

A third observation arises from the previous discussion, it is that nearly all requirements change due to the changes in the business/commercial, political or social environment. It is a significant problem for many of our systems today that soon after the system is developed and deployed it suffers from changes in the original environment it was developed for. This means that software development has to take evolution of the requirements into account during the system development, especially for complex systems.

[5] Whether it is due to requirements or has not been detected during verification in the inner loop.

4 Implications for Software Verification

The problem that the Program Verifier challenge addresses can be reduced to a mechanical procedure. When an error is detected then human intervention is still important in order to eliminate the class of errors detected, however it is systematic in nature and the rate of errors detected should diminish over time.

4.1 Observations

A key observation is that there is a trend towards the automation of code generation from simulation (and similar) models. The rise of automatic code generation is taking place for reasons quite separate from the Program Verifier grand challenge, but it could play a key role to the importance of the challenge. The grand challenge is then to provide Program Verifiers for commercial automated code generators from modelling languages into commercial languages. If source code generation and its formal verification can be automated then the problem of evolving requirements is mitigated if the cost of re-developing software and verifying it is significantly reduced. However the most important contribution of the Program Verifier challenge is that it could be scalable in a way that the elimination of requirements errors cannot be.

Consider the following simplistic, but useful, illustration. Suppose the elimination of requirement errors through the use formal techniques and human expertise resulted in a 50% saving in a development. Now consider the limited expertise available. For a number of development projects totalling £100, 000, 000, 000 (one hundred billion pounds), it would not be unreasonable to consider that only 1% could be addressed by the scarce human expertise available. This means that savings of 50% of one billion pounds could be achieved, a not inconsiderable sum of five hundred million pounds.

The scalability of the result of the Program Verifier challenge means that potentially all the projects totalling one hundred billion pounds could be addressed. Even a 10% saving would result in a saving of ten billion pounds.

4.2 An Experiment

In September 2002, QinetiQ agreed to run an experiment to replace 80 of the 200 functional modules of version 3R1 of the flight control law with auto-generated, autoproven SPARK Ada code in conjunction with BAE Systems, Rochester (BAES). The 80 selected modules represented various types of code including complexity, runtime (long/short), size and the applicability to the PowerPC benchmark employed by BAES. The manually developed versions of these modules represented approximately 8,500 lines of code of a total of approximately 18,000 lines of Ada source code (roughly 47% of the manually developed code and all non-blank and non-comment). Throughout the Malvern work, time measurements were taken for each part of the process and would be compared to industry norms and the equivalent from the original work undertaken by BAES.

Specifically the sample of 80 specifications of Ada procedures from the existing Flight Control Law source code were selected to provide a good representative mix of all types of code as follows:

- 18 were from an existing PowerPC benchmark, to give a complete path through the control law;
- 22 had varying runtimes, for the manually developed code, from short to long;
- 20 had a McCabe complexity measure between 1 and 40 inclusive, for the manually developed code;
- 20 had varying storage from small to large, for the manually developed code.

The baseline for the specification of these procedures was version 3R1 of the flight control laws. A Simulink representation of the Fortran specification, used by BAES, acted as the specification for the verification of the automatically generated code. The equivalence between the Fortran and its Simulink representation had been established during the post development verification exercise on behalf of the Eurofighter IPT. There were a number of limitations on the experiment that influenced results, however even taking these into account independent evaluation indicates a saving of 30% over the system development lifecycle [1].

An important aspect of the experiment was to compare costs to achieve the same outcome, to pass the same set of acceptance tests. This means that the objective of software verification was not to achieve a failure rate better than that of the conventional development, rather it was to achieve at least the same quantifiable failure rate but at significantly less cost.

4.3 Certification

Software aspects of certification are normally guided by development guidelines like DO-178B [5]. DO-178B identifies a set of software activities that can take place within the context of a broader engineering process, and requires an accomplishment summary [5, p.71] as part of the certification deliverables. The accomplishment summary is a document that demonstrates compliance of a system with the plan for software aspects of certification.

As part of this process, various forms of testing (for example unit testing, integration testing and regression testing) are undertaken. These testing processes are very important, but also very costly, leading to a desire to replace testing with more cost-effective processes, where practicable. In [6] the authors are concerned with how such technology substitution can be justified, and presents a template for an argument that can be used to justify substitutions. It also instantiates the argument for QinetiQ's proof technology (used for the software verification of Typhoon's flight control law software) and demonstrates how to argue for its safe substitution for testing in this context. Thus the use of software verification can be made on cost grounds while satisfying the objectives of current development guidelines such as DO-178B.

5 Conclusions

In order to achieve the potential benefits of the Program Verifier grand challenge a number of research issues must be addressed, the following constitute a minimum.

- A validated and rigorous measurement framework for comparing costs and benefits for development processes.
- The development of Program Verifiers for commercial language subsets with respect to commercially accepted modelling languages, this implies two supporting objectives:
    - language semantics for subsets of commercial languages (such as C and C++) that are mechanically checkable and sympathetic to program verification needs;
    - model language semantics that are mechanically checkable and sympathetic to program verification needs.
- Automated proof maintenance to support the evolution of the software.

Much work has already been done in these areas, but the products of these areas needs to be brought together and put into a coherent framework to support the Program Verifier Grand Challenge.

A repository to collate results in this area would be an important step. The repository should also be used to make a judgement on the fitness for purpose of products in these areas. For example there are many semantic models for languages, there are rather fewer that are mechanically verifiable and directly support software verification. Many aspects of the judgement will be scientifically based, however some, such as usefulness to industry, will be partly subjective. The repository would also provide a means of conducting scientific experiments to measure the effectiveness (through testing) of a program verifier and compare its cost against historical data in a meaningful way [2] as was done for the Typhoon experiment [1].

The popular prejudice is that formal methods add cost to add reliability, this paper challenges this view. There is little evidence to show that a program verifier will make a scientifically quantifiable improvement in reliability before deployment of a system (the evidence that the use of a program verifier led to greater reliability 10 or 30 years later is not useful). Scientific experiments to determine the cost of achieving a quantifiable reliability target, implicit in an acceptance test criteria, can be conducted with an appropriate measurement framework. A programme of work to drive down the costs of verification and validation for commercial models and languages is required. The trend is towards a set of automatic code generators from commercial modelling languages that lend themselves to automated verification i.e. a set of program verifiers.

Systems are becoming more complex increasing the chances that errors will occur at higher levels, but in order to address this we need secure foundations. Software verification is important because it provides a solid foundation for Formal Methods to be applied in the design and requirements phase to gain further benefits.

An interesting question that has been posed is to what extent is the evidence and conjectured value of program verification expressed in this paper influenced by the particular characteristics of the problem domain, implementation of control systems that derive their basis from control theory. The domain of control theory is populated by highly scientifically literate control engineers. This bias in the population of "users" of the verification technology leads the question of would "the observation that non-formal and formal techniques lead to the same level of assurance, albeit at

different costs, still hold true when transferred to other computer application domains were the underlying theory was less well developed, and/or the "users" less technically capable"? I believe the answer to this is that there is clearly a link between the dependability of the software system and the maturity of theory that supports that domain. However this is a question about the validity of the models developed in a particular domain, not the correctness of the software implementing those models. Further the evidence from BAES Rochester is that Control Engineers were not involved in the implementation. Normal Software Engineers following a rigorous development process implemented the flight control software. Experience from the development and verification of a Digital Engine Control Unit supports this. It therefore seems to be the case that where appropriate modeling languages are used and validated, the conjecture on the value of a program verifier should hold true.

The Systems Assurance Group has demonstrated the benefits of a program verifier in a narrow area for a particular modelling language. Requirements analysis of control law models by developing a Hoare logic for Simulink was based on the original verification work conducted by QinetiQ [3, 4]. This demonstrates that software verification supports the use of formal methods in requirements and design activities. In summary:

- Rather than basing argument for adoption of formal software verification on enhanced reliability we should aim to be significantly cheaper than conventional development for the same quantifiable outcomes.
- The application of Formal Methods is most powerful at requirements and design stage.
- The application of Formal Methods at the requirements stage is not scalable because it requires human interpretation –more experts are needed and people cost more than machines.
- Coding and verification can be automated – it is scalable and costs can be significantly reduced mitigating rapid requirements change.
- Software verification is still important and provides a platform for Formal Methods to be applied in the design and requirements phase.
- A repository would allow experiments to be conducted similar to the one conducted by the UK MOD, BAES and QinetiQ.
- In many areas requirements change quickly, a verifier makes the effort of re-work much less and mitigates the cost of change – this is Dependable Systems Evolution.

A short answer to the question posed by this paper is that scalability and significant cost reduction is where the true value in a program verifier lies.

References

1. Tudor, N., Adams, M., Clayton, P., O'Halloran, C.: Auto-coding/Auto-proving flight control software. In: Proceedings of IEEE 23rd Digital Avionics Systems Conference, Salt Lake City (2004)
2. Clark, G.D., Caseley, P.R., Powell, A.L., Murdoch, J.: Measurement of Safety Processes. In: Incose 2003, Washington, USA (2003)

3. Boulton, R.J., Gottliebsen, H., Hardy, R., Kelsy, T., Martin, U.: Design Verification for Control Engineering. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 21–35. Springer, Heidelberg (2004)
4. Boulton, R.J., Hardy, R., Martin, U.: A Hoare-Logic for Single-Input Single-Output Continuous-Time Control Systems. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 113–125. Springer, Heidelberg (2003)
5. RTCA Inc. and EUROCAE. DO-178B: Software Considerations in Airborne Systems and Equipment Certification (December 1992)
6. Galloway, A., Paige, R.F., Tudor, N.J., Weaver, R.A., Toyn, I., McDermid, J.: Proof Vs Testing in the context of safety standards. In: Proceedings of IEEE 24th Digital Avionics Systems Conference (2005)
7. Adams, M.M., Clayton, P.B.: CLawZ: Cost-Effective Formal Verification for Control Systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785. Springer, Heidelberg (2005)
8. Fauquemberque, J.-L., Kahn, G., Kubbat, W., Levedag, S., Lions, J.-L., Lubeck, L., Mazzini, L., Merle, D., O'Halloran, C.: ARIANE 5 Flight 501 Failure, Report by the Inquiry Board, ESA (1996)

Author Index