Electronic Notes in Theoretical Computer Science

Volume ???

3rd international Workshop on Systems Software Verification

SSV'08

Sydney, Australia

February 25-27, 2008

Guest Editors:

Ralf Huuck, Gerwin Klein, and Bastian Schlich

These are electronic pre-proceedings distributed at the workshop.
Please do not distribute.

Preface

This volume contains the proceedings of the 3rd International Workshop on Systems Software Verification (SSV) held in Sydney, Australia, February 25–27, 2008.

The purpose of SSV is to bring together researchers and developers from both academia and industry, who are facing real software and real problems to find real, applicable solutions. By "real" we mean problems such as time-to-market or reliability that the industry is facing and is trying to fix in software that is deployed in the market place. A real solution is one that is applicable to the problem in industry and not one that only applies to an abstract, academic toy version of it. SSV discusses software analysis/development techniques and tools, and serves as a platform to discuss open problems and future challenges in dealing with existing and upcoming system code.

This volume starts with the abstracts of two invited presentation, followed by 13 papers that were selected by the Programme Committee out of 24 submissions after an intense reviewing and discission phase. Each paper has been examined by at least 3 reviewers and we would like to thank the members of the Programme Committee as well as the external reviewers for their detailed and thorough investigation of each contribution.

The workshop programme comprised the presentation of the 13 accepted papers, four tool demonstrations given by

- Christof Efkemann and Helge Lding (University of Bremen, Germany),
- Ansgar Fehnker (NICTA, Australia),
- Ralf Huuck (NICTA, Australia),
- Bastian Schlich (RWTH Aachen, Germany),

and two invited talks presented by

- Kim Guldstrand Larsen (Aalborg University/CISS, Denmark) and
- Hongseok Yang (University of London, United Kingdom).

We would like to thank the the Programme Committee, the authors, and the workshop organizers for their contribution to the success of this 3rd International Workshop on Systems Software Verification. Finally, we are grateful for the generous support we received from National ICT Australia (NICTA) funded under the Australian Governments Backing Australias Ability program through the Department of Communications, Information Technology and the Arts (DCITA) and the Australian Research Council (ARC).

February 2008 Ralf Huuck, Gerwin Klein, and Bastian Schlich

Organization

Program Chair

Ralf Huuck	NICTA, Australia
Gerwin Klein	NICTA, Australia
Bastian Schlich	RWTH Aachen University, Germany

Program Committee

Andy Chou	Coverity, Inc., USA
Byron Cook	Microsoft Research Cambridge Lab, UK
Dennis Dams	Alcatel-Lucent, USA
Ansgar Fehnker	NICTA, Australia
David Hardin	Rockwell Collins, USA
Joost-Pieter Katoen	RWTH Aachen University, Germany
Thomas Kropf	Robert Bosch GmbH, Germany
Gerald Luettgen	University of York, UK
John Matthews	Galois, Inc., USA
Tobias Nipkow	Technical University of Munich, Germany
Michael Norrish	NICTA, Australia
Wolfgang Paul	Saarland University, Germany
Jan Peleska	University of Bremen, Germany
Alexander Pretschner	ETH Zurich, Switzerland
Kai Richter	Symtavision GmbH, Germany
Zhong Shao	Yale University, USA
Hendrik Tews	Radboud University Nijmegen, The Netherlands
Stavros Tripakis	Cadence Research Labs, USA
Carsten Weise	RWTH Aachen University, Germany

Local Organizers

Ralf Huuck	NICTA, Australia
Gerwin Klein	NICTA, Australia
Belinda Selig	NICTA, Australia

Referees

Rahul Agarwal
Josh Berdine
Kirsten Berkenktter
Andy Chou
Efkemann Christof
David Cock
Byron Cook
Dennis Dams
Ulan Degenbaev
Jonathan Ezekiel
Ansgar Fehnker
Sabine Fischer
Florian Haftmann
Tingting Han
Ulrich Hannemann
David Hardin
Ralf Huuck
Joost-Pieter Katoen
Gerwin Klein
Rafal Kolanski
Alexander Krauss
Thomas Kropf
Dirk Leinenbach
Gerald Lttgen
John Matthews
Laurent Mounier
Tobias Muehlberg
Tobias Nipkow
Thomas Noll
Michael Norrish
Wolfgang Paul
Jan Peleska
Alexander Pretschner
Kai Richter
Stefan Rieger
Norbert Schirmer
Bastian Schlich
Thomas Sewell
Zhong Shao
Hendrik Tews
Stavros Tripakis
Carsten Weise
Hongseok Yang

Contents

Kim Larsen (Invited Speaker)
Validation, Performance Analysis and Synthesis of Embedded Systems 1

Hongseok Yang (Invited Speaker)
On scalable Shape Analysis 2

Pär Emanuelsson and Ulf Nilsson
A Comparative Study of Industrial Static Analysis Tools 3

Dirk Leinenbach and Elena Petrova
Pervasive Compiler Verification - From Verified Programs to Verified Systems 20

Harvey Tuch
Structured Types and Separation Logic 37

Rafal Kolanski
A Logic for Virtual Memory 55

Hendrik Tews, Tjark Weber and Marcus Völp
A Formal Model of Memory Peculiarities for the Verification of Low-Level Operating-System Code 71

Christof Efkemann and Tobias Hartmann
Specification of Conditions for Error Diagnostics 88

Helge Löding and Jan Peleska
Symbolic and Abstract Interpretation for C/C++ Programs 104

Gerlind Herberich, Bastian Schlich, Carsten Weise, and Thomas Noll
Proving Correctness of an Efficient Abstraction for Interrupt Handling 120

Tom In der Rieden and Alexandra Tsyban
CVM - A Verified Framework for Microkernel Programmers 137

Artem Starostin and Alexandra Tsyban
Correct Microkernel Primitives 154

Paul Graunke
Verified Safety and Information Flow of a Block Device 170

Kirsten Berkenkötter
Reliable UML Models and Profiles 186

Tony Cant, Ben Long, Jim McCarthy, Brendan Mahony, and Kylie Williams
The HiVe Writer 202

Invited talk: Validation, Performance Analysis and Synthesis of Embedded Systems

Kim Larsen

Aalborg University, Denmark

Abstract

Within the upcoming European Joint Technology Initiative ARTEMIS as well as several national initiatives such as CISS (www.ciss.dk) and DaNES (http://www.danes.aau.dk/), model-driven development is a key to dealing with the increasing complexity of embedded systems, while reducing the time and cost to market. The use of models should permit early assessment of the functional correctness of a given design as well as requirements for resources (e.g. energy, memory, and bandwidth) and real-time and performance guarantees. Thus, there is a need for quantitative models allowing for timed, stochastic and hybrid phenomenas to be modeled an analysed.
UPPAAL and the branches CORA and TIGA provide an integrated tool environment for modelling, validation, verification and synthesis of real-time systems modelled as networks timed automata, extended with data types and user-defined functions. The talk will provide details on the expressive power of timed automata in relationship to embedded systems as well as details on the power and working of the UPPAAL verification engine.
In this talk we demonstrate how UPPAAL has been applied to the validation, performance analysis and synthesis of embedded control problems. The applications include so-called task graph scheduling and MP-SoC systems consisting of application software running under different RTOS on processors interconnected through an on-chip network. Also we show how CORA and TIGA has been used to synthesize optimal (e.g. wrt. energy or memory) scheduling strategies given applications, including Dynamic Voltage Scaling and a climate controller.

Keywords: UPPAAL, Timed Automata

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Invited talk: On Scalable Shape Analysis

Hongeseok Yang

Queen Mary University of London, UK

Abstract

Shape analysis is a precise form of pointer analysis, which can be used to verify deep properties of data structures such as whether or not they are cyclic, whether they are nested, etc. Shape analyses are also expensive, and the tremendous number of abstract states they generate is an impediment to their use in verification of sizeable programs. In this talk, I will describe the techniques for improving the scalability of shape analyses. With these techniques, we have improved our analysis that was able to handle programs of up to 1,000 lines, such that it can now analyze programs of up to 10,000 lines. Our experiments also show that the new analysis is precise. It identifies memory safety errors and memory leaks in several Windows and Linux device drivers and, after these bugs are fixed, it automatically proves integrity of pointer manipulation for these drivers.

This order of magnitude improvement in sizes of programs verified is obtained by combining several ideas. One is the local reasoning idea of separation logic, which reduces recomputation of analysis of procedure bodies, and which allows efficient transfer functions for primitive program statements. Another is an interprocedural analysis algorithm which aggressively discards intermediate states. The most important new technical contribution of the work is a new join (or widening) operator, which greatly reduces the number of abstract states used by the analysis while not greatly reducing precision; the join is also integrated with procedure summaries in an interprocedural analysis.

This is joint work with Oukseh Lee, Cristiano Calcagno, Dino Distefano and Peter O'Hearn.

Keywords: Separation Logic, Shape Analysis

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

A Comparative Study of Industrial Static Analysis Tools

Pär Emanuelsson[1,2]

Ericsson AB
Datalinjen 4
SE-583 30 Linköping, Sweden

Ulf Nilsson[1,3]

Dept. of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Abstract

Tools based on static analysis can be used to find defects in programs. Tools that do shallow analyses based on pattern matching have existed since the 1980's and although they can analyze large programs they have the drawback of producing a massive amount of warnings that have to be manually analyzed to see if they are real defects or not. Recent technology advances has brought forward tools that do deeper analyses that discover more defects and produce a limited amount of false warnings. These tools can still handle large industrial applications with millions lines of code. This article surveys the underlying supporting technology of three state-of-the-art static analysis tools. The survey relies on information in research articles and manuals, and includes the types of defects checked for (such as memory management, arithmetics, security vulnerabilities), soundness, value and aliasing analyses, incrementality and IDE integration. This survey is complemented by practical experiences from evaluations at the Ericsson telecom company.

Keywords: Static analysis, dataflow analysis, defects, security vulnerabilities.

1 Introduction

Almost all software contain *defects*. Some defects are found easily while others are never found, typically because they emerge seldom or not at all. Some defects that emerge relatively often even go unnoticed simply because they are not perceived as errors or are not sufficiently severe. Software defects may give rise to several types of *errors*, ranging from logical/functional ones (the program sometimes computes incorrect values) to runtime errors (the program typically crashes), or resource leaks (performance of the program degrades possibly until the program freezes or crashes).

[1] Thanks to Dejan Baca, Per Flodin, Fredrik Hansson, Per Karlsson, Leif Linderstam, and Johan Ringström from Ericsson for providing tool evaluation information.

[2] Email: par.emanuelsson@ericsson.com

[3] Email: ulfni@ida.liu.se

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Programs may also contain subtle security vulnerabilities that can be exploited by malicious attackers to gain control over computers.

Fixing defects that suddenly emerge can be extremely costly in particular if found at the end of the development cycle, or even worse, after deployment. Many simple defects in programs can be found by modern compilers, but the predominating method for finding defects is *testing*. Testing has the potential of finding most types of defects, however, testing is costly and no amount of testing will find all defects. Testing is also problematic because it can be applied only to executable code, i.e. rather late in the development process. Alternatives to testing, such as *dataflow analysis* and *formal verification*, have been known since the 1970s but have not gained widespread acceptance outside academia—that is, until recently; lately several commercial tools for detecting runtime error conditions at compile time have emerged. The tools build on *static analysis* [27] and can be used to find runtime errors as well as resource leaks and even some security vulnerabilities statically, i.e. without executing the code. This paper is a survey and comparison of three market leading static analysis tools in 2006/07: PolySpace Verifier, Coverity Prevent and Klocwork K7. The list is by no means exhaustive, and the list of competitors is steadily increasing, but the three tools represent state-of-the-art in the field at the moment.

The main objective of this study is (1) to identify significant static analysis functionality provided by the tools, but not addressed in a normal compiler, and (2) to survey the underlying supporting technology. The goal is not to provide a ranking of the tools; nor is it to provide a comprehensive survey of all functionality provided by the tools. Providing such a ranking is problematic for at least two reasons: Static analysis is generally only part of the functionality provided by the tool; for instance, Klocwork K7 supports both refactoring and software metrics which are not supported by the two other tools. Even if restricting attention only to static analysis functionality the tools provide largely non-overlapping functionality. Secondly, even when the tools seemingly provide the same functionality (e.g. detection of dereferencing of null pointers) the underlying technology is often not comparable; each tool typically finds defects which are not found by any of the other tools.

Studying the internals of commercial and proprietary tools is not without problems; in particular, it is virually impossible to get full information about technical solutions. However, *some* technical information is publicly available in manuals and white papers; some of the tools also originate from academic tools which have been extensively described in research journals and conference proceedings. While technical solutions may have changed somewhat since then, we believe that such information is still largely valid. We have also consulted representatives from all three providers with the purpose to validate our descriptions of the tools. Still it must be pointed out that the descriptions of suggested technical solutions is subject to a certain amount of guessing in some respects.

This technological survey is then complemented by a summary and some examples of tool evaluations at Ericsson.

The rest of the report is organized as follows: In Section 2 we define what we mean by the term static analysis and survey some elementary concepts and preconditions; in particular, the trade off between precision and analysis time. In Section

3 we summarize the basic functionality provided by the three tools—Coverity Prevent, Klocwork K7 and PolySpace Verifier/Desktop—focusing in particular on the support for the C and C++ programming languages. The section also surveys several industrial evaluations of the tools over time at Ericsson, in particular involving the products from Coverity and Klocwork. Section 4 contains conclusions.

2 Static analysis

Languages such as C and, to a lesser extent, C++ are designed primarily with efficiency and portability in mind [4], and therefore provide little support to avoid or to deal with runtime errors. For instance, there is no checking in C that read or write access to an array is within bounds, that dereferencing of a pointer variable is possible (that the variable is not null) or that type casting is well-defined. Such checks must therefore be enforced by the programmer. Alternatively we must make sure that the checks are not needed, i.e. guarantee that the error conditions will never occur in practice.

By the term *static analysis* we mean *automatic* methods to reason about runtime properties of program code without actually executing it. Properties that we consider include those which lead to premature termination or ill-defined results of the program, but precludes for instance purely syntactic properties such as syntax errors or simple type errors. [5] Nor does static analysis address errors involving the functional correctness of the software. Hence, static analysis can be used to check that the program execution is not prematurely aborted due to unexpected runtime events, but it does not guarantee that the program computes the correct result. While static analysis *can* be used to check for e.g. deadlock, timeliness or non-termination there are other, more specialized, techniques for checking such properties; although relying on similar principles. Static analysis should be contrasted with *dynamic analysis* which concerns analysis of programs based on their execution, and includes e.g. testing, performance monitoring, fault isolation and debugging.

Static analysis does not in general guarantee the absence of runtime errors, and while it can reduce the need for testing or even detect errors that in practice cannot be found by testing, it is not meant to replace testing.

The following is a non-exhaustive list of runtime problems that typically cannot be detected by traditional compilers and may be difficult to find by testing, but which can be found by static analysis:

- Improper resource management: Resource leaks of various kinds, e.g. dynamically allocated memory which is not freed, files, sockets etc. which are not properly deallocated when no longer used;
- Illegal operations: Division by zero, calling arithmetic functions with illegal values (e.g. non-positive values to logarithm), over- or underflow in arithmetic expres-

[4] Or so it is often claimed; in fact, even in ANSI/ISO Standard C there are many language constructs which are not semantically well-defined and which may lead to different behavior in different compilers.

[5] The borderline is not clear; some checks done by compilers, such as type checking in a statically typed language, are closer to runtime properties than syntactic ones. In fact, in a sufficiently rich type system some type checking must be done dynamically.

sions, addressing arrays out of bounds, dereferencing of null pointers, freeing already deallocated memory;

- Dead code and data: Code and data that cannot be reached or is not used. This may be only bad coding style, but may also signal logical errors or misspellings in the code;
- Incomplete code: This includes the use of uninitialized variables, functions with unspecified return values (due to e.g. missing return statements) and incomplete branching statements (e.g. missing cases in `switch` statements or missing `else` branches in conditional statements).

Other problems checked for by static analysis include non-termination, uncaught exceptions, race conditions etc.

In addition to finding errors, static analysis can also be used to produce more efficient code; in particular for "safe" languages like Java, where efficiency was not the primary objective. Many runtime tests carried out in Java programs can in practice be avoided given certain information about the runtime behavior. For instance, tests that array indices are not out-of-bounds can be omitted if we know that the value of the indices are limited to values in-bounds. Static analysis can provide such information.

Static analysis can also be used for type inference in untyped or weakly typed languages or type checking in languages with non-static type systems [21]. Finally static analysis can be used for debugging purposes (see e.g. [1]), for automatic test case generation (see e.g. [19]), for impact analysis (see e.g. [26]), intrusion detection (see e.g. [29]) and for software metrics (see e.g. [30]). However, in this paper we focus our attention on the use of static analysis for finding defects and software vulnerabilities which typically would not show up until the code is executed.

Most interesting properties checked by static analyses are *undecidable*, meaning that it is impossible, even in theory, to determine whether an arbitrary program exhibits the property or not. As a consequence static analyses are inherently *imprecise*—they typically infer that a property (e.g. a runtime error) *may* hold. This implies that

(i) if a program has a specific property, the analysis will usually only be able to infer that the program *may* have the property. In some special cases the analysis may also be able to infer that the program *does* have the property.

(ii) if the program does not have the property, there is a chance that (a) our analysis is actually able to infer this (i.e. the program *does not* have the property), but it may also happen that (b) the analysis infers that the program *may* have the property.

If the property checked for is a defect then we refer to case (ii)(b) as a *false positive*. Hence, if the analysis reports that a program may divide by zero we cannot tell in general whether it is a real problem (item (i)) or if it is a false positive (item (ii)(b)). The precision of the analysis determines how often false positives are reported. The more imprecise the analysis is, the more likely it is to generate false positives.

Unfortunately precision usually depends on analysis time. The more precise

the analysis is, the more resource consuming it is, and the longer it takes. Hence, precision must be traded for time of analysis. This is a very subtle trade-off—if the analysis is fast it is likely to report many false positives in which case the alarms cannot be trusted. On the other hand a very precise analysis is unlikely to terminate in reasonable time for large programs.

One way to avoid false positives is to filter the result of the analysis, removing potential errors which are unlikely (assuming some measure of likelihood). However, this may result in the removal of positives which are indeed defects. This is known as a *false negative*—an actual problem which is not reported. False negatives may occur for at least two other reasons. The first case is if the analysis is too optimistic, making unjustified assumptions about the effects of certain operations. For instance, not taking into account that `malloc` *may* return null. The other case which may result in false negatives is if the analysis is incomplete; not taking account of all possible execution paths in the program.

There are a number of well-established techniques that can be used to trade-off precision and analysis time. A *flow-sensitive* analysis takes account of the control flow graph of the program while a *flow-insensitive* analysis does not. A flow-sensitive analysis is usually more precise—it may infer that `x` and `y` may be aliased (only) after line 10, while a flow-insensitive analysis only infers that `x` and `y` may be aliased (anywhere within their scope). On the other hand, a flow-sensitive analysis is usually more time consuming.

A *path-sensitive* analysis considers only valid paths through the program. It takes account of values of variables and boolean expressions in conditionals and loops to prune execution branches which are not possible. A path-insensitive analysis takes into account all execution paths—even infeasible ones. Path-sensitivity usually implies higher precision but is usually more time consuming.

A *context-sensitive* analysis takes the context—e.g. global variables and actual parameters of a function call—into account when analyzing a function. This is also known as *inter-procedural* analysis in contrast to *intra-procedural* analysis which analyses a function without any assumptions about the context. Intra-procedural analyses are much faster but suffer from greater imprecision than inter-procedural analyses.

Path- and context-sensitivity rely on the ability to track possible values of program variables; for instance, if we do not know the values of the variables in the boolean expression of a conditional, then we do not know whether to take the `then`-branch or the `else`-branch. Such *value analysis* can be more or less sophisticated; it is common to restrict attention to intervals (e.g. `0 < x < 10`), but some approaches rely on more general relations between several variables (e.g. `x > y+z`). Another important issue is aliasing (see e.g. [14,28]); when using pointers or arrays the value of a variable can be modified by modifying the value of another variable. Without a careful value and aliasing analyses we will typically have large numbers of false positives, or one has do ungrounded, optimistic assumptions about the values of variables.

The undecidability of runtime properties implies that it is impossible to have an analysis which always finds all defects and produces no false positives. A framework for static analysis is said to be *sound* (or *conservative* or *safe*) if all defects

checked for are reported, i.e. there are no false negatives but there may be false positives. [6] Traditionally, most frameworks for static analysis have aimed for soundness while trying to avoid excessive reporting of false positives (e.g. the products from PolySpace). However, most commercial systems today (e.g. Coverity Prevent and Klocwork K7) are not sound (i.e. they will not find all actual defects) and also typically produce false positives.

It is sometimes claimed that static analysis can be applied to incomplete code (individual files and/or procedures). While there is some truth to this, the quality of such an analysis may be arbitrarily bad. For instance, if the analysis does not know how a procedure or subprogram in existing code is called from outside it must, to be sound, assume that the procedure is called in an arbitrary way, thus analyzing executions that probably cannot occur when the missing code is added. This is likely to lead to false positives. Similarly incomplete code may contain a call to a procedure which is not available, either because it is not yet written, or it is a proprietary library function. Such incomplete code *can* be analyzed but is also likely to lead to a large number of false positives and/or false negatives depending on if the analysis makes pessimistic or optimistic assumptions about the missing code.

On the positive side, it is often not necessary to provide complete code for missing functions or function calls. It is often sufficient to provide a stub or a top-level function that mimics the *effects* of the properties checked for.

The tools studied in this report adopt different approaches to deal with incomplete code and incremental analysis when only some code has been modified (as discussed in the next section).

3 A comparison of the tools

Shallow static analysis tools based on pattern matching such as FlexeLint [17] have existed since the late 1980s. Lately several sophisticated industrial-strength static analysis tools have emerged. In this report we study tools from three of the main providers—PolySpace, Coverity and Klocwork. There are several other static analysis tools around, including PREfix/PREfast from Microsoft [3], Astree [7], which are not as widely available. A tool which has existed for some years but not until recently has become commercially available is CodeSonar from Grammatech, founded by Tim Teitelbaum and Tom Reps, which is similar in style and ambition level to Coverity Prevent and Klocwork K7, see [18]. Even if we focus here on tools intended for global and "deep" (=semantic) analysis of code, more lightweight tools like FlexeLint may still be useful in more interactive use and for local analysis.

There are also dynamic tools that aim for discovering some of the kinds of defects as the static analysis tools do. For example Insure++ [22] and Rational Purify [24]

[6] Soundness can be used in two completely different senses depending on if the focus is on the reporting of defects or on properties of executions. In the former (less common) sense soundness would mean that all positives are indeed defects, i.e. there are no false positives. However, the more common sense, and the one used here, is that soundness refers to the assumptions made about the possible executions. Even if there is only a small likelihood that a variable takes on a certain value (e.g. `x=0`) we do not exclude that possibility. Hence if the analysis infers that `X` *may* be zero in an expression `1/x`, there is a possibility that there will be a runtime error; otherwise not. This is why a sound analysis may actually result in false positives, but no false negatives.

detect memory corruption errors.

A rough summary of major features of the three systems studied here can be found in Table 1. Such a table is by necessity incomplete and simplistic and in the following sub-section we elaborate on the most important differences and similarities. A more thorough exposition of the tools can be found in the full version of the paper, see [16].

3.1 Functionality provided

While all three tools have much functionality in common, there are noticeable differences; in particular when comparing PolySpace Verifier [15,23] against Coverity Prevent [10,11] and Klocwork K7 [20]. The primary aim of all three tools obviously is to find real defects, but in doing so any tool will also produce some false positives (i.e. false alarms). While Coverity and Klocwork are prepared to sacrifice finding all bugs in favor of reducing the number of false positives, PolySpace is not; as a consequence the former two will in general produce relatively few false positives but will also typically have some false negatives (defects which are not reported). It is almost impossible to quantify the rate of false negatives/positives; Coverity claims that approximately 20 to 30 per cent of the defects reported are false positives. Klocwork K7 seems to produce a higher rate of false positives, but stays in approximately the same league. However, the rate of false positives obviously depends on the quality of the code. The rate of false negatives is even more difficult to estimate, since it depends even more on the quality of the code. (Obviously there will be *no* false negatives if the code is already free of defects.) According to Coverity the rate of defect reports is typically around 1 defect per 1-2 KLoC.

PolySpace, on the other hand, does in general mark a great deal of code in orange color which means that it *may* contain a defect, as opposed to code that is green (no defects), red (definite defect) or grey (dead code). If orange code is considered a potential defect then PolySpace Verifier produces a high rate of false positives. However, this is a somewhat unfair comparison; while Coverity and Klocwork do not even give the developer the opportunity to inspect all potential defects, PolySpace provides that opportunity and provides instead a methodology in which the developer can systematically inspect orange code and classify it either as correct or faulty. In other words, Coverity and Klocwork are likely to "find some bugs", but provide no guarantees—the rest of the code may contain defects which are not even reported by the tool. PolySpace on the other hand can provide guarantees—if all code is green (or grey) it is *known* not to contain any bugs (wrt the properties checked for, that is). On the other hand it may be hard to eliminate all orange code.

All three tools rely at least partly on inter-procedural analyses, but the ambition level varies significantly. PolySpace uses the most advanced technical solution where relationships between variables are approximated by convex polyhedra [8] and all approximations are sound—that is, no execution sequences are forgotten, but some impossible execution paths may be analyzed due to the approximations made. Coverity Prevent and Klocwork K7 account only of interval ranges of variables in combination with "simple" relationships between variables in a local context with the main purpose to prune some infeasible execution paths, but do not do as well as

PolySpace. Global variables and nontrivial aliasing are not accounted for or treated only in a restricted way. As a consequence neither Coverity nor Klocwork take all possible behaviors into account which is one source of false negatives. It is somewhat unclear how Coverity Prevent and Klocwork K7 compare with each other, but impression is that the former does a more accurate analysis.

Another consequence of the restricted tracking of arithmetic values of variables in Coverity Prevent and Klocwork K7 is that the products are not suitable for detecting arithmetic defects, such as over- and underflows or illegal operations like division by zero. The products did not even provide arithmetic checkers at the time of the study. PolySpace on the other hand does provide several arithmetic checkers, setting it apart from the others.

While PolySpace is the only tool that provides arithmetic checkers, it is also the only one among the three which does not provide any checkers for resource leaks; in particular there is no support for discovering defects in dynamic management (allocation and deallocation) of memory. As a consequence there are also no checkers e.g. for "use-after-free". This lack can perhaps be explained by PolySpace's focus on the embedded systems market, involving safety or life critical applications where no dynamic allocation of memory is possible or allowed.

While PolySpace appears to be aiming primarily for the embedded systems market, Klocwork and Coverity have targeted in particular networked systems and applications as witnessed, for instance, by a range of security checkers. Klocwork and Coverity address essentially the same sort of security issues ranging from simple checks that critical system calls are not used inappropriately to more sophisticated analyses involving buffer overruns (which is also supported by PolySpace) and the potential use of so-called tainted (untrusted) data. The focus on networked application also explains the support for analyzing resource leaks since dynamic management of resources such as sockets, streams and memory is an integral part of most networked applications.

Coverity supports incremental analysis of a whole system, where only parts have been changed since last analysis. Results of an analysis are saved and reused in subsequent analyses. An automatic impact analysis is done to detect and, if necessary, re-analyze other parts of the code affected indirectly by the change. Such an incremental analysis may take significantly less time than analyzing the whole system from scratch. With the other tools analysis of the whole system has to be redone. All of the tools provide the possibility to analyze a single file. However such an analysis will be much more shallow than analyzing a whole system where complete paths of execution can be analyzed.

Both Klocwork and Coverity provide means for writing user defined checkers and integrating them with the analysis tools, see e.g. [9,4]. However, the APIs are non-trivial and writing new, non-trivial checkers is both cumbersome and error prone. There are no explicit guidelines for writing correct checkers and no documented support for manipulation of abstract values (e.g. interval constraints). There is also no support for reusing the results of other checkers. Termination of the checker is another issue which may be problematic for users not familiar with the mathematical foundations of static analysis, see e.g. [6,27].

All three tools support analysis of the C programming language and C++. At

Table 1
Summary of features of Coverity Prevent, Klocwork K7 and PolySpace Verifier

Functionality	Coverity	KlocWork	PolySpace
Coding style	No	Some	No
Buffer overrun	Yes	Yes	Yes
Arithmetic over/underflow	No	No	Yes
Illegal shift operations	No	No	Yes
Undefined arithmetic operations	No	No	Yes
Bad return value	Yes	Yes	Yes
Memory/resource leaks	Yes	Yes	No
Use after free	Yes	Yes	No
Uninitialized variables	Yes	Yes	Yes
Size mismatch	Yes	Yes	Yes
Stack use	Yes	No	No
Dead code/data	Yes	Yes	Yes (code)
Null pointer dereference	Yes	Yes	Yes
STL checkers	Some	Some	No?
Uncaught exceptions	Beta (C++)	No	No
User assertions	No	No	Yes
Function pointers	No	No	Yes
Nontermination	No	No	Yes
Concurrency	Lock order	No	Shared data
Tainted data	Yes	Yes	No
Time-of-check Time-of-use	Yes	Yes	No
Unsafe system calls	Yes	Yes	No
MISRA support	No	No	Yes
Extensible	Yes	Some	No
Incremental analysis	Yes	No	No
False positives	Few	Few	Many
False negatives	Yes	Yes	No
Software metrics	No	Yes	No
Language support	C/C++	C/C++/Java	C/C++/Ada

the initial time of this study only Klocwork supported analysis of Java but Coverity was announcing a new version of Prevent with support for Java. Only PolySpace supported analysis of Ada. Klocwork was the only provider which claimed to handle mixed language applications (C/C++/Java).

The downside of PolySpace's sophisticated mechanisms for tracking variable values is that the tool cannot deal automatically with very large code bases without manual partitioning of the code. While Coverity Prevent and Klocwork K7 are able to analyze millions of lines of code off-the-shelf and overnight, PolySpace seems to reach the complexity barrier already at around 50 KLoC with the default settings.

On the other hand PolySpace advocates analyzing code in a modular fashion. Analysis time is typically not linear in the number of lines of code—analyzing 10 modules of 100 KLoC is typically orders of magnitude faster than analyzing a single program consisting of 1,000 KLoC. However this typically involves human intervention and well-defined interfaces (which may be beneficial for other quality reasons...)

On the more exotic side Coverity provides a checker for *stack use*. It is unclear how useful this is since there is no uniform way of allocating stack memory in different compilers. Klocwork is claimed to provide similar functionality but in a separate tool. PolySpace set themselves aside from the others by providing checkers for non termination, both of functions and loops. Again it is unclear how useful such checkers are considering the great amount of research done on *dedicated* algorithms for proving termination of programs (see e.g. [13,2]). Coverity has a checker for uncaught exceptions in C++ which was still a beta release. PolySpace provides a useful feature in their support for writing general *assertions* in the code. Such assertions are useful both for writing stubs and may also be used for proving partial correctness also of functional properties; see [25].

None of the tools provide very sophisticated support for dealing with concurrency. Klocwork currently provides no support at all. Coverity is able to detect some cases of mismatched locks but does not take concurrency into account during analysis of concurrent threads. The only tool which provides more substantial support is PolySpace which is able to detect shared data and whether that data is protected or not.

Both Coverity and Klocwork have developed lightweight versions of their tools aimed for frequent analysis during development. These have been integrated with Eclipse IDEs. However the defect databases for Coverity and Klocwork have not been integrated into Eclipse IDEs or TPTP. PolySpace has integrated with the Rhapsody UML tool to provide a UML static analysis tool. It analyzes generated code and links back references to the UML model to point out where defects have been detected. Besides that PolySpace has its general C++ level advantages with a sound analysis (no false negatives) and presumably problems with analyzing large code bases (larger than 50-100 KLoC)—a restriction which should be more severe in the UML situation compared to hand-coded C++.

3.2 Experiences at Ericsson

A number of independent evaluations of static analysis tools were performed by development groups at Ericsson. Coverity was evaluated by several groups. Klocwork has also been subject to evaluations but not quite as many. There was an attempt to use PolySpace for one of the smallest applications, but the evaluation was not successful; the tool has either presented no results within reasonable time (a couple of days' execution) or the results were too weak to be of use (too much orange code to analyze). We do not know if this was due to the tool itself or to the actual configuration of the evaluations. It would have been valuable to compare results from PolySpace, which is sound, to those of Klocwork and Coverity. Perhaps that would give some hint on the false negative rate in Klocwork and Coverity.

Some general experiences from use of Coverity and Klocwork were:

- The tools are easy to install and get going. The development environment is easy to adapt and no incompatible changes in tools or processes are needed.
- The tools are able to find bugs that would hardly be found otherwise.
- It is possible to analyze even large applications with several million lines of code and the time it takes is comparable to build time.
- Even for large applications the false positive rate is manageable.
- Several users had expected the tools to find more defects and defects that were more severe. On the other hand, several users were surprised that the tools found bugs even in applications that had been tested for a long time. There might be a difference in what users find reasonable to expect from these tools. There might also be large differences in what users classify as a false positive, a bug or a severe bug.
- It is acceptable to use tools with a high false positive rate (such as FlexeLint) if the tool is introduced in the beginning of development and then used continuously.
- It is unacceptable to use tools with a high false positive rate if the product is large and the tool is introduced late in the development process.
- Many of the defects found could not cause a crash in the system as it was defined and used at the moment. However if the system would be only slightly changed or the usage was changed the defect could cause a serious crash. Therefore these problems should be fixed anyway.
- Even if the tools look for the same categories of defects, for instance memory leaks, addressing out of array bounds etc, the defects found in a given category by one tool can be quite different from those found by another tool.
- Handling of third party libraries can make a big difference to analysis results. Declarations for commercial libraries that come with the analysis tool can make the analysis of own code more precise. If source for the library is available defects in the library can be uncovered, which may be as important to the quality of the whole application as the own code.
- There are several aspects of the tools that are important when making a tool selection that has not been a part of the comparison in this paper; such as pricing, ease of use, integration in IDEs, other functionality, interactiveness etc.

Below follows some more specific results from some of the evaluations. We do not publish exact numbers of code sizes and found bugs etc for confidentiality reasons since some of the applications are commercial products in use.

Evaluation 1 (Coverity and FlexeLint): The chosen application had been thoroughly tested, both with manually designed tests and systematic tests that were generated from descriptions. FlexeLint was applied and produced roughly 1,200,000 defect reports. The defects could be reduced to about 1,000 with a great deal of analysis and following filtering work. These then had to be manually analyzed. Coverity was applied to the same piece of code and found about 40 defects; there were very few false positives and some real bugs. The users appreciated the low false positive rate. The opinion was that the defects would hardly have been found

by regular testing.

The users had expected Coverity to find more defects. It was believed that there should be more bugs to be found by static analysis techniques. It was not known if this was the price paid for the low false positive rate or if the analyzed application actually contained only a few defects. The users also expected Coverity to find more severe defects. Many of the findings were not really defects, but code that simply should be removed, such as declarations of variables that were never used. Other defects highlighted situations that could not really occur since the code was used in a restricted way not known to the analysis tool.

Evaluation 2 (Coverity): A large application was analyzed with Coverity. Part of the code had been previously analyzed with FlexeLint. The application had been extensively tested.

Coverity was perceived both as easy to install and use, and no modifications to existing development environment was needed. The error reports from the analysis were classified as follows

- 55 per cent were no real defects but perceived only as poor style,
- 2 per cent were false positives,
- 38 per cent were considered real bugs, and 1 per cent were considered severe.

The users appreciated that a fair number of defects were found although the code had already been thoroughly tested.

Evaluation 3 (Coverity and Klocwork): An old version of an application that was known to have some memory leaks was analyzed using Coverity and Klocwork.

In total Klocwork reported 32 defects including 10 false positives and Coverity reported 16 defects including 1 false positive. Only three defects were common to both tools! Hence Klocwork found more defects, but also had a larger false positive rate. Although the tools looked for similar defects the ones actually found were largely specific to each tool. This suggests that each of the tools fail in finding many defects.

Looking at only the memory leaks the results were similar. Klocwork reported 12 defects of which 8 were false, totalling 4 real defects and Coverity reported 7 defects all of which were true defects. None of the tools found any of the known memory leaks.

Evaluation 4 (Coverity and Klocwork): Old versions of two C++ products were analyzed with Coverity and Klocwork. Trouble reports for defects that had been detected by testing were available. One purpose was to compare how many faults each of the tools found. Another purpose was to estimate how many of the faults discovered in testing were found by the static analysis tools.

Coverity found significantly more faults and also had significantly less false positives than Klocwork. One of the major reasons for this was the handling of third party libraries. Coverity analyzed the existing source code for the libraries and found many faults in third party code! Klocwork did not analyze this code and

hence did not find any of these faults. Besides that the analysis of the libraries that Coverity did resulted in fewer false positives in the application code since it could be derived that certain scenarios could not occur.

The time of analyses was about the same as build time for both tools—i.e. is good enough for overnight batch runs but not for daily, interactive use during development.

Both tools lacked integration with CM tool Clearcase, the source code had to be copied into the repository of the analysis tools. There was no way to do inspection of analysis results from an IDE, but the reviews had to be done in the GUI of the analysis tools.

Coverity was preferred by the C++ developers. It had incremental analysis that would save time and it could easily analyze and report on single components.

Although the main part of the evaluation was on old code some studies were done on programs during the development. The development code had more warnings and most of them were real faults; most of these were believed to have been found during function test. It had been anticipated that more faults would be found in low level components, but these components proved to be stable and only a few defects were discovered. More faults were however found in high level components with more frequent changes.

Evaluation 5 (Coverity, Klocwork and CodePro): A Java product with known bugs was analyzed. A beta version of Coverity Prevent with Java analysis capabilities was used. None of the known bugs were found by the tools. Coverity found more real faults and had far less false positives than Klocwork. For Coverity one third of the warnings were real bugs.

Klocwork generated many warnings; 7 times the number of warnings of Coverity. The missing analysis of the third party library seemed to be the major reason. However, Klocwork does a ranking of the potential defects and when only the four most severe levels of warnings were considered the results were much better—there were few false positives.

CodePro Analytix (developed and marketed by Instantiations) is a tool aimed for analysis during development. It is integrated into the Eclipse IDE and the results of an analysis cannot be persistently saved, but only exist during the development session with the IDE. The analysis is not as deep as that of Coverity or Clockwork, but is faster and can easily be done interactively during development. The tool generates a great deal of false positives, but these can be kept at a tolerable level by choosing an appropriate set of analysis rules. No detailed analysis was done of the number of faults and if they were real faults or not.

In this evaluation there was a large difference in the number of warnings generated, Coverity 92 warnings, Klocwork 658 warnings (in the top four severities 19), CodePro 8,000 warnings (with all rules activated).

4 Conclusions

Static analysis tools for detection of runtime defects and security vulnerabilities can roughly be categorized as follows

- **String and pattern matching approaches**: Tools in this category rely mainly on syntactic pattern matching techniques; the analysis is typically path- and context-insensitive. Analyses are therefore shallow, taking little account of semantic information except user annotations, if present. Tools typically generate large volumes of false positives as well as false negatives. Tools (often derivatives of the lint program) have been around for many years, e.g. FlexeLint, PC-Lint and Splint. Since the analysis is shallow it is possible to analyze very large programs, but due to the high rate of false positives an overwhelming amount of post-processing may be needed. These tools are in our opinion more useful for providing almost immediate feedback in interactive use and in combination with user annotations.
- **Unsound dataflow analyses**: This category of tools which have emerged recently rely on semantic information; not just syntactic pattern matching. Tools are typically path- and context-sensitive but the precision is limited so in practice the tools have to analyze also many impossible paths or make more-or-less justified guesses what paths are (im-)possible. This implies that analyses are unsound. Aliasing analysis is usually only partly implemented, and tracking of possible variable values is limited; global variables are sometimes not tracked at all. A main objective of the tools, represented e.g. by Coverity Prevent and Klocwork K7, is to reduce the number of false positives and to allow for analysis of very large code bases. The low rate of false positives (typically 20–30 per cent in Coverity Prevent) is achieved by a combination of a unsound analysis and filtering of the error reports. The downside is the presence of false negatives. It is impossible to quantify the rate since it depends very much on the quality of the code, but in several evaluations Coverity and Klocwork find largely disjoint sets of defects. This category of tools provide no guarantees—the error reports may or may not be real defects (it has to be checked by the user), and code which is not complained upon may still be defective. However, the tools will typically find *some* bugs which are hard to find by other techniques.
- **Sound dataflow analyses**: Tools in this category are typically path- and context-sensitive. However, imprecision may lead to analysis of some infeasible paths. They typically have sophisticated mechanisms to track aliasing and relationships between variables including global ones. The main difficulty is to avoid excessive generation of false positives by being as precise as possible while analysis time scales. The only commercial system that we are aware of which has taken this route is PolySpace Verifier/Desktop. The great advantage of a sound analysis is that it gives some guarantees: if the tool does not complain about some piece of code (the code is green in PolySpace jargon) then that piece of code must be free of the defects checked for.

There is a forth category of tools which we have not discussed here—namely tools based on *model checking* techniques [5]. Model checking, much like static analysis, facilitates traversal and analysis of all reachable states of a system (e.g. a piece of software), but in addition to allowing for checking of runtime properties, model checking facilitates checking of functional properties (e.g. safety properties) and also so-called temporal properties (liveness, fairness and real-time properties). There

are commercial tools for model checking hardware systems, but because of efficiency issues there are not yet serious commercial competitors for software model checking.

It is clear that the efficiency and quality of static analysis tools have reached a maturity level were static analysis is not only becoming a viable complement to software testing but is in fact a required step in the quality assurance of certain types of applications. There are many examples where static analysis has discovered serious defects and vulnerabilities that would have been very hard to find using ordinary testing; the most striking example is perhaps the Scan Project [12] which is a collaboration between Stanford and Coverity that started in March, 2006 and has reported on more than 7,000 defects in a large number of open-source projects (e.g. Apache, Firebird, FreeBSD/Linux, Samba) during the first 18 months.

However, there is still substantial room for improvement. Sound static analysis approaches, such as that of PolySpace, still cannot deal well with very large code bases without manual intervention and they produce a large number of false positives even with very advanced approximation techniques to avoid loss of precision. Unsound tools, on the other hand, such as those from Coverity and Klocwork do scale well, albeit not to the level of interactive use. The number of false positives is surprisingly low and clearly at an acceptable level. The price to be paid is that they are not sound, and hence, provide no guarantees: they may (and most likely will) find some bugs, possibly serious ones. But the absence of error reports from such a tool only means that the *tool* was unable to find any potential defects. As witnessed in the evaluations different unsound tools tend to find largely disjoint defects and are also known not to find known defects. Hence, analyzed code is likely to contain dormant bugs which can only be found by a sound analysis.

Most of the evaluations of the tools have been carried out on more or less mature code. We believe that to fully ripe the benefits of the tools they should not be used only at the end of the development process (after testing and/or after using e.g. FlexeLint), but should probably be used throughout the development process. However, the requirements on the tools are quite different at an early stage compared to at acceptance testing. Some vendors "solve" the problem by providing different tools, such as PolySpace Desktop and PolySpace Verifier. However, we rather advocate giving the user means of fine-tuning the behavior of the analysis engine. A user of the tools today has very limited control over precision and the rate of false positives and false negatives—there are typically a few levels of precision available, but the user is basically in the hands of the tools. It would be desirable for the user to have better control over precision of the analyses. There should for example be a mechanism to fine-tune the effort spent on deriving value ranges of variables and the effort spent on aliasing analysis. For some users and in certain situations it would be acceptable to spend five times more analysis time in order to detect more defects. Before an important release it could be desirable to spend much more time than on the day to day analysis runs. In code under development one can possibly live with some false negatives and non-optimal precision as long as the tool "finds some bugs". As the code develops one can improve the precision and decrease the rate of false positives and negatives; in particular in an incremental tool such as Coverity Prevent. Similarly it would be desirable to have some mechanism to control the aggressiveness of filtering of error reports.

References

[1] Ball, T. and S. Rajamani, *The SLAM Project: Debugging System Software via Static Analysis*, ACM SIGPLAN Notices **37** (2002), pp. 1–3.

[2] Ben-Amram, A. M. and C. S. Lee, *Program Termination Analysis In Polynomial Time*, ACM Trans. Program. Lang. Syst. **29** (2007).

[3] Bush, W., J. Pincus and D. Sielaff, *A Static Analyzer For Finding Dynamic Programming Errors*, Software, Practice and Experience **30** (2000), pp. 775–802.

[4] Chelf, B., D. Engler and S. Hallem, *How to Write System-specific, Static Checkers in Metal*, in: *PASTE '02: Proc. 2002 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering* (2002), pp. 51–60.

[5] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, Cambridge, MA, USA, 1999.

[6] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction Or Approximation of Fixpoints*, in: *Conf. Record of the Fourth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (1977), pp. 238–252.

[7] Cousot, P., R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, *The ASTRÉE Analyser*, in: M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, Lecture Notes in Computer Science **3444** (2005), pp. 21–30.

[8] Cousot, P. and N. Halbwachs, *Automatic Discovery of Linear Restraints Among Variables of a Program*, in: *Conf. Record of the Fifth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (1978), pp. 84–97.

[9] Coverity Inc., "Coverity Extend™ User's Manual (2.4)," (2006).

[10] Coverity Inc., *Coverity Prevent™: Static Source Code Analysis for C and C++* (2006), product information.

[11] Coverity Inc., "Coverity Prevent™ User's Manual 2.4," (2006).

[12] Coverity Inc., *The Scan Ladder* (2007), URL: http://scan.coverity.com.

[13] Dershowitz, N. and Z. Manna, *Proving Termination With Multiset Orderings*, Commun. ACM **22** (1979), pp. 465–476.

[14] Deutsch, A., *Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting*, in: *Proc. PLDI* (1994), pp. 230–241.

[15] Deutsch, A., *Static Verification of Dynamic Properties*, White paper, PolySpace Technologies Inc (2003).

[16] Emanuelsson, P. and U.Nilsson, *A Comparative Study of Industrial Static Analysis Tools (Extended Version)*, Technical Reports in Computer and Information Science no 2008:3, Linköping University Electronic Press (2008), URL: http://www.ep.liu.se/ea/trcis/2008/003/.

[17] Gimpel Software, "PC-lint/FlexeLint," (1999), URL: http://www.gimpel.com/lintinfo.htm.

[18] GrammaTech Inc., *Overview of GrammaTech Static Analysis Technology* (2007), white paper.

[19] King, J., *Symbolic Execution and Program Testing*, Comm. ACM **19** (1976), pp. 385–394.

[20] Klocwork Inc., *Detected Defects and Supported Metrics* (2005), K7 product documentation.

[21] Palsberg, J. and M. Schwartzbach, *Object-Oriented Type Inference*, in: *Conf Proc Object-Oriented Programming Systems, Languages, And Applications (OOPSLA '91)* (1991), pp. 146–161.

[22] Parasoft Inc., *Automating C/C++ Runtime Error Detection With Parasoft Insure++* (2006), white paper.

[23] PolySpace Technologies, "PolySpace for C Documentation," (2004).

[24] Rational Software, *Purify: Fast Detection of Memory Leaks and Access Errors*, White paper (1999).

[25] Rosenblum, D. S., *A Practical Approach to Programming With Assertions*, IEEE Trans. Softw. Eng. **21** (1995), pp. 19–31.

[26] Ryder, B. and F. Tip, *Change Impact Analysis For Object-Oriented Programs*, in: *Proc. of 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis For Software Tools And Engineering (PASTE '01)* (2001), pp. 46–53.

[27] Schwartzbach, M., *Lecture Notes on Static Analysis* (2006), BICS, Univ. Aarhus, URL: http://www.brics.dk/~mis/static.pdf.

[28] Steensgaard, B., *Points-to Analysis in Almost Linear Time*, in: *ACM POPL*, 1996, pp. 32–41.

[29] Wagner, D. and D. Dean, *Intrusion Detection via Static Analysis*, in: *Proc. of 2001 IEEE Symp. on Security and Privacy (SP'01)* (2001), pp. 156–168.

[30] Wagner, T., V. Maverick, S. Graham and M. Harrison, *Accurate Static Estimators For Program Optimization*, in: *Proc. of ACM SIGPLAN 1994 Conf. on Programming Language Design And Implementation (PLDI '94)* (1994), pp. 85–96.

Pervasive Compiler Verification – From Verified Programs to Verified Systems

Dirk Leinenbach[1,2,3]

German Research Center for Artificial Intelligence (DFKI)
P.O. Box 15 11 50
66041 Saarbrücken, Germany

Elena Petrova[1,4]

Saarland University, Computer Science Dept.
P.O. Box 15 11 50
66041 Saarbrücken, Germany

Abstract

We report in this paper on the formal verification of a simple compiler for the C-like programming language $C0$. The compiler correctness proof meets the special requirements of pervasive system verification and allows to transfer correctness properties from the $C0$ layer to the assembler and hardware layers. The compiler verification is split into two parts: the correctness of the compiling specification (which can be translated to executable ML code via Isabelle's code generator) and the correctness of a $C0$ implementation of this specification. We also sketch a method to solve the boot strap problem, i.e., how to obtain a trustworthy binary of the $C0$ compiler from its $C0$ implementation. Ultimately, this allows to prove pervasively the correctness of compiled $C0$ programs in the *real* system.

Keywords: Compiler Verification, Theorem Proving, System Verification, HOL, Hoare Logic

1 Introduction

The Verisoft project aims at the pervasive formal verification of computer systems comprising hardware (the verified VAMP processor [7,12] and devices [1,18]), system software [15], and applications [5]. 'Pervasive' means to prove a single, integrated correctness theorem for the whole system instead of verifying separate properties for each layer without justification that they *formally* fit together (cf. [30]).

Except for very small parts of the system level software, software in Verisoft is implemented in the C-like programming language $C0$. This language has been

[1] Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project (http://www.verisoft.de) under grant 01 IS C38.

[2] Work supported by DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

[3] Email: Dirk.Leinenbach@dfki.de

[4] Email: petrova@wjpserver.cs.uni-sb.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

designed to be expressive enough to allow implementation of low-level software while—at the same time—being 'neat' to allow for efficient formal verification of medium-sized $C0$ applications. However, pervasive verification does not stop at the $C0$ level. To allow execution of verified programs on the *real* hardware they must be 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 system verification when using a high-level programming language. Furthermore, the formulation of the compiler correctness statement has to be adequate for pervasive verification [24].

In order to bridge the gap between verified software and verified hardware, we have defined a *compiling specification* for a $C0$ compiler in Isabelle / HOL [35] and additionally *implemented* the compiler in $C0$. Both the compiling specification and its implementation have been formally verified [23,38]. For the latter we have shown using a $C0$ verification environment [40] that it produces the same list of assembler instructions as specified by the compiling specification. For the former we have verified a small-step simulation theorem, which states that the original $C0$ program and the compiled code behave equivalently. This theorem respects resource restrictions (e.g., bounded memory size) of the target machine and permits to discharge them at the $C0$ level. That the theorem is formulated in a small-step manner allows to argue about interleaving and non-terminating computations.

This paper is supposed to give an overview of the compiler verification efforts in Verisoft. For more details and precise formal definitions see [23,38].

1.1 Requirements Analysis and Related Work

Compiler verification is a well established field [13]. There are correctness proofs covering issues from simple expression translation in [27] to compilers with optimizations in [8,25]. Also, different source languages are considered: from toy languages to subsets of C [25] and Java [41] or the Java virtual machine [22].

In the Verifix project [14], impressive work concerning correct compilers has been done. In [44], the authors present an elegant theory for the translation of intermediate languages to machine languages; the work was partially formalized in the PVS theorem prover. The implementation of a compiler for ComLisp (a subset of Common Lisp) was verified on the machine code level by a manual check [16].

Recently, Leroy et al. have formally verified an optimizing two-step translation from Clight (a subset of C) first to the intermediate language C Minor and then to PowerPC assembler [8,25]. The proof in the Coq proof assistant is based on big-step semantics of the source and target languages. An executable compiler was obtained by automatic (unverified) extraction from the Coq specification.

However, a compiler correctness theorem to be used for pervasive system verification has to meet extra requirements. We highlight the most important ones.

Language Model

C0 is a sequential language and even the target machine is a uni-processor architecture. So, sequential reasoning, big-step semantics, and classical Hoare logics seem to be adequate. But interleaving and non-terminating system software as well

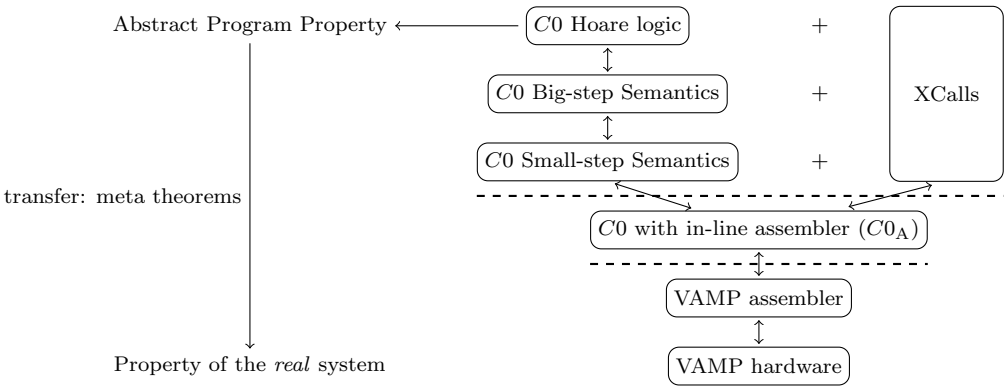

Fig. 1. Semantics Layers in Verisoft

as interrupt driven devices demand a concurrent model [2,18]. Small-step semantics and a small-step compiler correctness theorem are appropriate to handle this.

Compiler correctness proofs w.r.t. small steps semantics exist on paper [26,34]. But the proofs are usually carried out 'big step style' by a straightforward induction over the syntax tree. This works only for terminating programs. In our context it is *much* more comfortable to work with a compiler correctness statement in the form of a small-step simulation theorem as it has been done for a back-end in [44].

The Verisoft project uses several semantical layers to base reasoning on the right abstraction level [2]. This increases efficiency—when using the more abstract layers—while still allowing formulation and verification of detailed, concrete properties on the lower layers. Figure 1 depicts this stack. Results from the higher layers have to be formally transferred to the lower layers using meta theorems. Finally, this yields a single correctness theorem for the complete system. To support $C0$ programs which invoke in-line assembler code in the Hoare logic, we formalize the effect of the in-line assembler parts axiomatically using so-called *XCalls* [2]. Their implementation has to be plugged in at the level of the $C0_{\mathrm{A}}$ semantics which combines $C0$ with in-line assembler.

Pervasiveness

In [9] the specification of an optimizing compiler back-end from the SSA intermediate language has been formally verified. However, the machine model used there is not the language of a realistic processor and hence the work does not suffice to bridge the gap between software and hardware for pervasive verification. On the other hand, the work from [43] describes a framework for modeling the semantics of expression evaluation including non-determinism in the evaluation order. In the context of pervasive verification, such complicated languages are not desirable as they make correctness proofs of larger programs infeasible.

Pervasive verification has to handle resource restrictions on the target machine. Our compiler correctness theorem incorporates these restrictions and allows to discharge them at the C0 rather than at the assembler level which simplifies reasoning and increases productivity. The small-step character of our simulation theorem allows to easily argue about resource restrictions also for intermediate states.

The famous CLI project [6] resulted in a stack of verified components including a

compiler specification. The produced collection of verified programs has mostly been done in low-level languages. Recently, Zhong Shao [33] presented very nice logics for the assembler level verification of different kinds of low-level software. However, to allow for the *efficient* verification of medium-sized applications we have to use a high-level implementation language.

Early papers consider only verification of a compiling specification rather than verification of its implementation, although in [11,29] the authors already pointed out the necessity of the implementation correctness proof. Later, Goerigk et al. added a new aspect of compiler correctness, namely the *bootstrapping problem*, i.e., generation of the first trustworthy executable of a verified compiler [16].

Integration of Solutions

As pointed out above, there are many additional challenges for compiler verification due to pervasive verification. Some of them have been solved (in isolation) in a similar or even more general way in other work. However, in the context of pervasive verification an essential part of the verification effort has to be invested in the combination of the individual solutions into a single framework. In addition to the impressive work of the CLI stack project [6], early work from Joyce [21] discusses problems imposed by the formal combination of a verified compiler with verified hardware. To the best of our knowledge, the work presented in this paper is the first which integrates all the separate solutions into a single framework that provably [2] meets the needs of pervasive verification of complex systems.

1.2 Outline

The remainder of this paper is structured as follows. In Section 2, we introduce the $C0$ language and sketch its small-step semantics. We present a simulation theorem for the compiling specification in Section 3 and a correctness proof for the compiler implementation in Section 4. The section about the correct compiler implementation contains a sketch of our approach to solve the bootstrap problem. We conclude in Section 5 and discuss some future work.

2 The $C0$ Language

Semantics of the full C language are complex [17,36,37] and the use of all features of C leads to an error-prone programming style [31]. In contrast, formal verification of programs is easier and more efficient for programming languages with concise semantics. Verisoft uses the C-like imperative language $C0$ which has sufficient features to implement all system and application software in Verisoft while still allowing for efficient verification of programs with several thousand lines of code.

$C0$ has several limitations compared to standard C [20]; we list the most important ones. Side effects in expressions are not allowed, which forbids in particular function calls as subexpressions and requires a special function call statement. Pointers are typed and must not point to local variables or to functions; void pointers and pointer arithmetic are not supported. Arrays have to be of fixed size and

$$\begin{aligned} ty \;=\; & \mathit{Bool}_\mathrm{T} \mid \mathit{Int}_\mathrm{T} \mid \mathit{Char}_\mathrm{T} \mid \mathit{Unsigned}_\mathrm{T} \\ & \mid \mathit{Str}_\mathrm{T}(\mathbb{S} \times \mathit{ty}\ \mathit{list}) \mid \mathit{Arr}_\mathrm{T}(\mathbb{N}, \mathit{ty}) \mid \mathit{Ptr}_\mathrm{T}(\mathbb{S}) \mid \mathit{Null}_\mathrm{T} \end{aligned}$$

Fig. 2. Data Type *ty* for *C*0 Types

$$\begin{aligned} \mathit{expr} \;=\; & \mathit{Lit}(\mathit{lit}) \mid \mathit{Var}(\mathbb{S}) \mid \mathit{Arr}(\mathit{expr}, \mathit{expr}) \mid \mathit{Str}(\mathit{expr}, \mathbb{S}) \\ & \mid \mathit{UnOp}(\mathit{unop}, \mathit{expr}) \mid \mathit{BinOp}(\mathit{binop}, \mathit{expr}, \mathit{expr}) \\ & \mid \mathit{LazyBinOp}(\mathit{lazyop}, \mathit{expr}, \mathit{expr}) \\ & \mid \mathit{AddrOf}(\mathit{expr}) \mid \mathit{Deref}(\mathit{expr}) \end{aligned}$$

Fig. 3. Data Type *expr* for *C*0 Expressions

are represented by a separate type in *C*0. Low-level data types (unions or bit fields) and control flow statements (switch, goto, long jumps) are not supported.

*C*0 supports four *basic* types: booleans, 32-bit signed integers, 32-bit natural numbers, and 8-bit signed integers. Pointers, fixed size arrays, and structures are supported as *aggregate* types (cf. Figure 2). Pointer types do not directly include the type to which the point; instead, we use an additional indirection via *type names*. This allows the definition of self-referencing pointer types (e.g., a list component type whose 'next' field is a pointer to the component type). The mapping from type names to types is handled via a so-called type name environment (cf. Section 2.1). Observe, that there exists a special type for null pointer constants. *Elementary* types comprise basic types and pointers.

Variable names and literals are expressions. If e and i are expressions and cn is a component name, then array access $e[i]$, access to structure components $e.cn$, dereferencing $*e$, and the 'address-of' operator $\&e$ are also expressions. Additionally, *C*0 supports the usual unary and binary operators. In Figure 3, we give a formal definition of the data type *expr* which models *C*0 expressions in Isabelle.

*C*0 statements are modeled in Isabelle via the data type *stmt* (cf. Figure 4). Observe, that statements of a *C*0 program are annotated with unique identifiers of type *sid* (which is isomorphic to the natural numbers). These identifiers allow us to map statements occurring in the dynamic program rest to the original statements in the function table of a *C*0 program and to determine the function they belong to and their relation to other statements of the program.

In the following, let s and e (with arbitrary subscripts) denote statements and expressions. Besides sequential composition $\mathit{comp}(s_1, s_2)$, while loops $\mathit{while}(e, s)$, conditional statements $\mathit{if}(e, s_1, s_2)$, and the empty statement *skip*, *C*0 supports the following statements.

Assignments come in two flavors. *Normal* assignments $\mathit{ass}(e_l, e_r)$ copy the value of one expression to another. Unlike standard C, *C*0 supports assignments of arbitrary aggregate types.[5] *Complex* assignments $\mathit{ass}_\mathrm{C}(e_l, l_c)$, which assign a *complex*

[5] In addition to the rather restricted assignments of structures in C90, the C99 standard supports assignments like `x = (struct s){.n1 = e1, n2 = e2};`. However, C restricts this kind of assignments to initializers.

$$
\begin{aligned}
stmt \;=\; & skip \mid comp(stmt, stmt) \\
& \mid ass(expr, expr, sid) \mid ass_{\mathrm{C}}(expr, lit_{\mathrm{c}}, sid) \mid new(expr, \mathbb{S}, sid) \\
& \mid return(expr, sid) \mid if(expr, stmt, stmt, sid) \\
& \mid while(expr, stmt, sid) \mid asm(asm\ list, sid) \\
& \mid scall(\mathbb{S}, \mathbb{S}, expr\ list, sid) \mid xcall(\mathbb{S}, expr\ list, expr\ list, sid)
\end{aligned}
$$

Fig. 4. Data Type *stmt* for *C*0 Statements

literal l_c to an expression, are needed to initialize variables of aggregate types in a *single* step. This is required for the equivalence proof to the Hoare logic [40]. The left side of complex assignments is a normal expression of some aggregate type and the right side is a literal of the same type. Observe, that complex literals are only supported in this special case and must not be used inside normal expressions.

Dynamic allocation of zero-initialized heap memory for a type t is supported via $new(e, t)$ which assigns a pointer to the newly allocated memory region to the left side expression e. Observe, that *C*0 does not support explicit deallocation. Instead, a garbage collector will be used to deallocate unreachable parts of the heap in user applications. [6] The implementation correctness of a copying garbage collector for *C*0 has already been formally verified but is not yet integrated into the compiler correctness proof.

Function calls to a function f with parameters e_1 to e_n are represented by $scall(x, f, e_1, \ldots, e_n)$. Because *C*0 expressions must not have side effects, function calls are not supported as subexpressions. Instead, the return value of the function will be copied implicitly to variable x. Return from functions is handled by $return(e)$.

In the remainder of this paper we will often use the shorthand notation $r; s; t$ instead of $comp(r, comp(s, t))$ for consecutive statements r, s, and t.

2.1 C0 Small-step Semantics

*C*0 programs are represented in Isabelle by a symbol table *gst* for the global variables, a type name environment *te*, and a function table *ft*. The symbol table is a list of variable names together with their 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 symbol table for the function's parameters, a symbol table for the local variables, the function's return type, and a statement representing the body of the function.

Configurations

Configurations c of the *C*0 small-step semantics consist of two components: the program rest $c.pr :: stmt$ and the memory configuration $c.mem$. The program rest stores those statements which still have to be executed. It is initialized with the

[6] The operating system kernel of the Verisoft project [19,15] does only allocate a fixed amount of memory at startup. Thus, garbage collection is not necessary and the collector is deactivated for the kernel.

body of the 'main' function and grows / shrinks during program execution. A program has terminated when $c.pr = skip$.

The memory configuration is a triple consisting of a global memory frame $c.mem.gm :: frame$, a stack of local memory frames $c.mem.lm :: (gvar \times frame)\ list$, and a memory frame for heap variables $c.mem.hm :: frame$. Each memory frame m consists of a symbol table $m.st$ which lists the variables of the frame and of a content function $m.ct :: \mathbb{N} \to mcell$ which maps addresses (natural numbers) 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. Each local memory frame stores additionally a so-called *g-variable* which encodes the memory location where the function's result has to be stored.

Generalized Variables

Generalized variables (short g-variables) are a structural way of referring to memory objects. Pointers in the $C0$ small-step semantics are represented using g-variables. There are three base cases for g-variables: global variables of name x are represented by $gvar_{\mathrm{gm}}(x)$, local variables x in the i-th local memory frame by $gvar_{\mathrm{lm}}(i, x)$, and nameless heap variables with index i by $gvar_{\mathrm{hm}}(i)$. The inductive case defines g-variables for structure and array access. If g is a g-variable of structure type then a component $g' = gvar_{\mathrm{str}}(g, n)$ of name n is also a g-variable. If g is a g-variable of array type then its i-th element $g' = gvar_{\mathrm{arr}}(g, i)$ is also a g-variable. In these two cases, g' is called a sub g-variable of g.

We inductively define the set of *reachable* g-variables: a g-variable g is reachable iff (i) g is a global or local g-variable, (ii) another reachable pointer g-variable points to g, or (iii) g is a sub g-variable of a reachable g-variable.

Expression Evaluation and Transition Function

The value of expressions e–remember that $C0$ expressions are side effect free–and g-variables g in configuration c is computed via $va(c, e)$ and $va(c, g)$, respectively. It is represented as a sequence of memory cells. The transition function δ_{C0} computes for a given $C0$ configuration c the next configuration c'. If a runtime error (e.g., division by zero) occurs, the functions returns the special error state $\perp$ which it will never leave. We define $C0$ computations by repeated application of the transition function: we start in an initial configuration c^0 and define inductively $c^{i+1} = \delta_{\mathrm{C0}}(c^i)$.

For later reference, we highlight some parts of the definition of the new program rest $c'.pr$. Let the old program rest start with statement s, i.e., $c.pr = s; r$. In most cases s is simply executed and the new program rest is set to $c'.pr = r$. In three cases the length of the program rest can grow. (i) If $s = while(e, s')$ and $va(c, e) = true$ then the new program rest is $c'.pr = s'; s; r$. (ii) If $s = if(e, s_1, s_2)$ then the new program rest is $c'.pr = s_1; r$ or $c'.pr = s_2; r$. (iii) If s is a function call to some function f with body b then the new program rest is $c'.pr = b; r$.

An Invariant on Program Rests

We prove an invariant about program rests of the $C0$ small-step semantics which will be used in the correctness proof for the compiling specification in Section 3:

each statement s in the program rest of a computation for some program p, except for return statements, is always followed by some statically determined *successor* statement $succ(p, s)$.

To formalize this invariant we need additional definitions. Observe, that we model partial functions in Isabelle with an option type; here, we hide this formalism and represent undefined values by the special symbol $\perp$.

We denote by $s2l :: stmt \rightarrow stmt\ list$ a function which flattens a statement tree spanned by skip and compound statements into a list of statements as follows:

$$s2l(s) = \begin{cases} [\,] & \text{if } s = skip \\ s2l(s_1) \circ s2l(s_2) & \text{if } s = comp(s_1, s_2)\ . \\ [s] & \text{otherwise} \end{cases}$$

Let p be a $C0$ program and fb the function body which contains statement s in the function table of p. We define the *parent statement* of s in program p in the following way.

$$pa(p, s) = \begin{cases} \perp & \text{if } s \in s2l(fb), \text{ i.e., if } s \text{ is a top-level statement} \\ s' & \text{if } \exists s' \in p.\ s' = while(e, lb) \wedge s \in s2l(lb) \\ s' & \text{if } \exists s' \in p.\ s' = if(e, s_1, s_2) \wedge (s \in s2l(s_1) \vee s \in s2l(s_2)) \end{cases}$$

By induction we define the i-th parent statement by $pa^0(p, s) = s$ and $pa^{i+1}(p, s) = pa^i(p, pa(p, s))$. We define the *environment* of statements s, i.e., the list of statements in the basic block which s belongs to.

$$env(p, s) = \begin{cases} s2l(s_1) & \text{if } pa(p, s) = if(e, s_1, s_2) \wedge s \in s2l(s_1) \\ s2l(s_2) & \text{if } pa(p, s) = if(e, s_1, s_2) \wedge s \in s2l(s_2) \\ s2l(lb) & \text{if } pa(p, s) = while(e, lb) \\ s2l(fb) & \text{otherwise, i.e., if } s \text{ is a top-level statement} \end{cases}$$

For a statement s we define its *direct successor* $succ_{\mathrm{d}}(p, s)$ to be the next statement in the environment of s. The direct successor is undefined if s is the last statement in $env(p, s)$. Finally, if s is not the last statement of a function body (in this case it would be a return statement), we recursively define its *successor* $succ(p, s)$.

$$succ(p, s) = \begin{cases} succ_{\mathrm{d}}(p, s) & \text{if } succ_{\mathrm{d}}(p, s) \neq \perp \\ pa(p, s) & \text{if } succ_{\mathrm{d}}(p, s) = \perp \text{ and } pa(p, s) \text{ is a while loop} \\ succ(p, pa(p, s)) & \text{otherwise} \end{cases}$$

In the following, we will always argue in the context of a fixed $C0$ program; thus, we will mostly omit the first parameter p of the above definitions.

Theorem 2.1 (Invariant on Program Rests) *If $s \in s2l(c^i.pr)$ for some step number i of a $C0$ computation and s is not a return statement then the next statement in $s2l(c^i.pr)$ is the successor statement of s, i.e., s is always followed by its successor statement.*

Proof. This theorem depends follows from the fact that the program rest of $C0$ programs only changes in a certain way. We prove it by induction on the step number i.

For all statement trees which are literally copied from the function table, the invariant holds by definition of $succ(s)$. This proves the induction base, where the program rest consists only of the body of the main function. For the induction step let the program rest be $c^i.pr = s; r$. We do a case distinction on the statement s, which will be executed in the next step.

If s is an assignment, a while with *false* condition, a return, or a new statement, s is simply consumed ($c^{i+1}.pr = r$) and the invariant obviously holds.

If $s = while(e, lb)$ and $va(c^i, e) = true$ we have $c^{i+1}.pr = lb; while(e, lb); r$. Because lb is part of the function table, the invariant holds for this part of the new program rest. The other part of the program rest remains unchanged. The crucial point is to prove that the while statement is the correct successor for the last statement s' of the loop body, formally: $succ(s') = while(e, lb)$. This follows from the second case of the definition of $succ(s)$.

If $s = if(e, s_1, s_2)$ the new program rest is $c^{i+1}.pr = s_1; r$ or $c^{i+1}.pr = s_2; r$, depending on the value of e. Let for both cases s' denote the last statement in the corresponding branch s_1 or s_2. We have to show that s' is followed by $succ(s')$ in the new program rest. By the third case of the definition of $succ$, we know that $succ(s') = succ(s)$. Therefore, we can conclude with help of the induction hypothesis that s' is followed by the same statement which followed the conditional s in the original program rest (in all cases this is the first statement in statement list r).

If s is a function call of some function f with body fb the new program rest is $c^{i+1}.pr = fb; r$. For r, the invariant still holds by induction hypothesis; for fb, it holds by definition of $succ$ because of fb being a sub tree of the function table. The interesting case is again the crossing from fb to r. However, in this special case there is nothing to show because the last statement in the function body is a return and the invariant does not state anything for return statements. □

3 Correctness of the Compiling Specification

The code generation algorithm of the $C0$ compiler is quite simple. 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. We denote code generation of the compiling specification for statements s and expressions e in this paper by $code_{\mathrm{s}}(s)$ and $code_{\mathrm{s}}(e)$; analogously, we denote the code generated by the implementation by $code_{\mathrm{i}}(s)$. With $cad(s)$ we denote the start address of the code which has been generated for statement s and with $ead(s)$ the address of the first instruction behind this code. As an example we present the code generation template for loops in Figure 5.

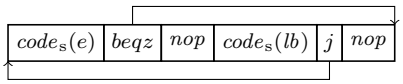

Fig. 5. Code Generation Template for Loops: $code_s(while(e, lb))$

3.1 Simulation Relation

We define a simulation relation between configurations c of the $C0$ machine and configurations d of the VAMP assembler machine. The latter are composed of two program counters $d.pc$ and $d.dpc$ implementing the delayed branch mechanism (see [7,32]), a word addressed memory $d.m$, and a general purpose register file $d.gpr$.

The set of valid g-variables of a $C0$ machine changes with *new* statements, function calls, and returns, and garbage collectors may change the allocated base address of heap g-variables. Thus, the simulation relation is parametrized with the current allocation function *alloc* which maps g-variables to their allocated base address in the VAMP assembler machine.

The simulation relation $consis(c, alloc, d)$ states that the VAMP configuration d encodes the $C0$ configuration c via the allocation function *alloc*. It comprises *control consistency* $consis_c(c, d)$ and *data consistency* $consis_d(c, alloc, d)$. Control consistency states that the VAMP's program counters point to the code of the first statement in the current program rest: $d.dpc = cad(hd(c.pr))$ and $d.pc = d.dpc + 4$. Data consistency is a conjunction of the following predicates.

Code consistency $consis_{code}(c, d)$ requires that the compiled code of the $C0$ program is stored at address 0 of the VAMP machine; this forbids self-modification.

Value consistency $consis_v(c, alloc, d)$ requires for all *reachable* g-variables g of basic type that $C0$ and VAMP machine store the same value: $d.m(alloc(g)) = va(c, g)$. For reachable pointer g-variables p which point to some g-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, i.e., $d.m(alloc(p)) = alloc(g)$. This defines a subgraph isomorphism between the reachable portions of the heaps of the $C0$ machine and the VAMP machine.

Stack consistency $consis_s(c, d)$ is a predicate on the implementation of the run time stack and the content of some special registers. Informally, it states that the first three words of each frame in the VAMP machine store the return address, i.e., where the code for final return statement jumps to, the destination address for the function's result, and a pointer to the previous frame. Additionally, we require that the return addresses in the VAMP agree with the control flow in the program rest of the $C0$ machine. Formally: that for all $i + j = |c.mem.lm|$ the return address stored in the j-th stack frame matches the address of the statement which follows the i-th return statement in the program rest.

3.2 Simulation Theorem

Essentially, the main theorem about the compiling specification 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 (cf. Figure 6). In reality, the theorem requires several additional preconditions.

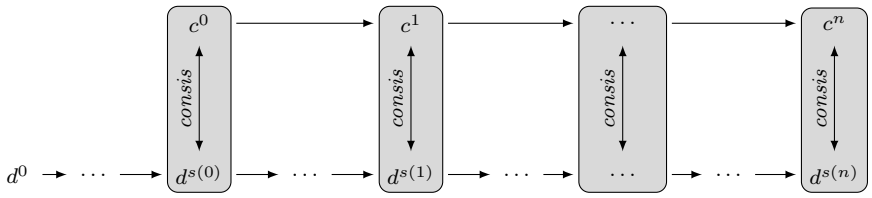

Fig. 6. Small-step Compiler Simulation Theorem

Theorem 3.1 (Simulation Theorem) *Let p be a C0 program, c^0 the corresponding initial configuration of the C0 machine, and d some well-formed initial assembler configuration which contains the compiled code $code_s(p)$ at address 0. Then, it holds for all steps i of the C0 machine executing program p that there exists an assembler step number $s(i)$ and an allocation function $alloc^i$ such that the C0 machine after i steps is consistent with the assembler machine after $s(i)$ steps.*

However, this is only true of the following requirements are fulfilled.[7]

- *The program p has to be translatable for our compiler: $p \in xltbl_{prog}$. Basically, this requires that the compiled code is not too big for the target machine, that jump distances fit into the immediate operands of the corresponding VAMP instructions, and that expression evaluation does not require too much registers to store intermediate results.*
- *We must not reach an error state up to step i of the C0 computation: $c^i \neq \bot$.*
- *There must not be a stack overflow up to step i of the C0 computation: $\forall j \leq i : \neg ovfl_{stack}(c^j)$.*

Formally, the theorem is stated as

$$\begin{aligned} &\forall i : p \in xltbl_{prog} \wedge c^i \neq \bot \wedge \forall j \leq i : \neg ovfl_{stack}(c^j) \\ &\quad \Longrightarrow \exists s(i), alloc^i : consis(c^i, alloc^i, d^{s(i)}). \end{aligned}$$

Proof. We prove this theorem by induction on i. For the induction start $i = 0$ we mainly have to show that the initialization part of $code_s(p)$ works correctly. The induction step from i to $i+1$ is proved by a case distinction over the first statement s in the program rest $c^i.pr = s; r$. We cannot present all cases here but concentrate on one interesting detail of the proof which comes from the fact that we prove a *small-step* simulation theorem.

Assume, that the program rest of the next configuration c^{i+1} starts with some statement s'. For control consistency, we have to show that the program counters eventually point to $cad(s')$. For the three cases in which the program rest grows (cf. Section 2.1), this proof is relatively easy because the correctness arguments are *local* regarding the statement s to be executed in step i. For return statements the proof follows immediately from stack consistency, which guarantees that the return addresses on the stack are correct.

For the remaining cases, s is completely consumed in the next step and we have $s' = r$. Thus, we have to show that we eventually reach the start of $code_s(r)$ where r—the new head of the program rest—is by Theorem 2.1 the successor statement

[7] Due to space restrictions we do not formally define the requirements in this paper. For details see [23].

of s. However, it is not guaranteed that $code_s(r)$ directly follows $code_s(s)$ in the compiled program; instead some *control code* might be placed between $code_s(s)$ and $code_s(r)$ (cf. Figure 7).

For example, consider s being the last statement in the if-branch s_1 of the conditional statement. There, $code_s(s)$ is followed by a jump instruction which skips the else-branch s_2 and this jump does not belong to $code_s(s_1)$ although it is indispensable to ensure control consistency after execution of s_1. In this case, the proof of control consistency is not *local* w.r.t. s but depends on the code for the conditional statement; even more, the jump instruction behind the code for the loop body also needs to be considered. However, in configuration c^i the conditional statement is no longer present in the program rest and we cannot easily argue about the correctness of the jump instruction. Instead we outsource the correctness proof of the control code into the following lemma. □

The first requirement of the previous theorem, i.e., that the program is translatable, is formulated in the executable subset of Isabelle's specification language and can be easily checked once and for all for a given $C0$ program using Isabelle's ML code generator. The other two requirements argue about runtime properties of the program; in the Verisoft scenario, they follow from the functional correctness proof of the program to be compiled.

Lemma 3.2 (Control Code Correctness) *Let the program counter of assembler configuration d^i point directly behind the last instruction of $code_s(s)$, where s is not a return statement. Then, it holds that after a certain number t of assembler steps we reach a configuration where the program counter points to the first instruction of the successor statement of s and the memory has not been changed.*

$$d^i.dpc = ead(s) \wedge d^i.pc = d^i.dpc + 4$$
$$\Longrightarrow \exists t : d^{i+t}.dpc = cad(succ(s)) \wedge d^{i+t}.pc = d^{i+t}.dpc + 4 \wedge d^{i+t}.m = d^i.m$$

Proof. We prove this theorem by induction—following structurally the definition of $succ(s)$. If s is not the last statement in a loop body or in the branch of some conditional, then there is no control code behind $code_s(s)$ and we are done.

If s is the last statement in the body of some loop $while(e, lb)$, we know from the definition of $succ$ that $while(e, lb)$ itself is the successor of s. By a proof, which is *local* w.r.t. the while loop, we can show that the distance of the jump instruction behind the loop body is correct and we finally have $d^{i+t}.dpc = cad(while(e, lb)) = cad(succ(s))$.

The most complicated case is when s is the last statement in a branch of some conditional statement. First, we show—using a simple auxiliary lemma—that the control code behind the conditional branch correctly jumps behind the code of the conditional statement, i.e., that we reach $cad(pa(s))$. Then, we apply the induction hypothesis to show that we finally reach $cad(succ(pa(s))) = cad(succ(s))$. □

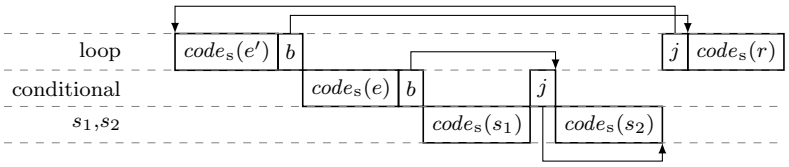

Fig. 7. Fragmented code generation for $code_s(while(e', if(e, s_1, s_2)); r)$

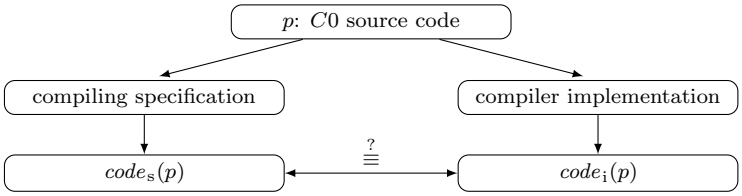

Fig. 8. Correctness of the Compiler Implementation

4 Implementation Correctness

For pervasive verification, it is not sufficient to have a verified compiling specification. Additionally, we need a verified compiler implementation in $C0$ which allows us (after boot strapping, cf. Section 4.4) to *execute* a verified compiler binary on the target platform. In addition to Theorem 3.1, it suffices to show that the compiler implementation produces the same code as the compiling specification (cf. Figure 8).

We have restricted the verification of the compiler implementation in Verisoft to the implementation of the code generation algorithm which consists of roughly 1.500 lines of $C0$ code in about 60 procedures. Due to limited project resources, parsing and I/O operations have not been verified. The verified compiler core is embedded into an unverified front-end written in C/C++, which parses a $C0$ input program, checks its syntactical correctness, and produces a syntax tree, which is then being fed into the compiler core. The verified core translates the $C0$ syntax tree into a list of VAMP assembler instructions, which is output by an unverified I/O routine.

The compiling specification works in one pass: offsets for relative jumps are determined on the fly via functions which compute solely the size of the generated code. In contrast, the compiler implementation in Section 4 uses two-pass compilation. Jump distances for relative jumps are left out in the first pass and filled in with correct values in a second pass when the position of all jump destinations is known.

4.1 Verification Environment

The compiler implementation has been verified in the $C0$ verification environment [40] which is based on a Hoare logic with an automatic verification condition generator (VCG) and allows to prove both partial and total correctness. The VCG automatically applies Hoare rules to a Hoare triple $\{P\}c\{Q\}$ by computing the weakest precondition WP for c, Q, and user-provided invariants for loops. After the program c is completely eliminated, the goal $P \longrightarrow WP$ is to be shown interactively in Isabelle.

The heap model features split heaps for every type (following ideas from [10]),

which gives separation of heap structures of different types for free. Additionally, the verification environment embeds $C0$ expressions *shallowly* into HOL to increase productivity. Due to the shallow embedding, the range of elementary types is—in contrast to the $C0$ small-step semantics—not bounded. Thus, expressions have to be annotated with so-called *guards* to allow the transfer of properties from the Hoare logic layer to the lower layers. Validity of such guards, which are generated automatically, implies the absence of run-time errors caused by over- or underflow.

4.2 Correctness Theorem

We formulate pre- and postconditions of the Hoare triples using so-called *abstraction relations*, which state the correspondence between the current state variables and abstract HOL types [28]. Abstraction relations have to be defined for all relevant data structures of the compiler implementation. Absence of pointers in the specification language results in very different representation of objects and, hence, makes abstraction relations and verification more complex.

As we do not prove the correctness of the front-end, we assume that the initial state σ of the compiler core contains a syntax tree of the input program and, analogously, the final state τ encodes the compiled instruction sequence. Formally, this is stated by the two top-level abstraction relations $C0_{\text{prog}}(\sigma, p)$ which states that σ encodes the $C0$ program p and $ASM_{\text{code}}(\tau, l)$ which states that τ encodes a list l of VAMP instructions. Using these abstraction relations we can formulate the top-level correctness theorem for the compiler implementation.

Theorem 4.1 *Let p be a $C0$ program, cimpl the $C0$ function which implements the code generation, and σ the initial state. Then, after executing cimpl, the final state τ encodes exactly that list of VAMP instructions which is specified by the compiling specification via $code_s(p)$. Formally, this is stated by the following Hoare triple.*

$$\{C0_{prog}(\sigma, p)\}\ \mathit{Call}\ \mathit{cimpl}();\ \{ASM_{code}(\tau, code_s(p))\}$$

4.3 Verification Issues

We highlight some of the key verification issues for the compiler implementation (besides code size). One of these follows from the implementation and specification being written in an imperative and a functional programming language, respectively. Thus, the correct implementation of recursive functions by while loops is an issue. Additionally, the recursion directions often differ; for example, in the implementation lists are traversed from head to tail and the specification exploits natural recursion with the last list element as induction base. Another example is the code generation for complex literals, where mutually recursive functions in the specification are implemented by a combination of recursive functions and loops.

In some cases, a *single* function in the specification is implemented by a combination of *several* $C0$ functions. One interesting example is the equivalence of the two-pass translation in the compiler implementation with the single-pass recursive function in the specification. Such cases require the introduction of additional intermediate states and predicates which allow to connect the different implementation

functions until we can finally prove their equivalence with the single specification function.

4.4 Boot Strapping

To solve the bootstrap problem [16], i.e., to obtain a trustworthy binary of the $C0$ compiler, it is not sufficient to verify the code generation algorithm or the $C0$ implementation of the compiler. In Verisoft, we follow two different ways how to get a trustworthy executable from the compiler implementation. First, B. Finkbeiner's group is currently applying translation validation techniques [39] to show that a binary compiler which has been generated by an untrusted bootstrap compiler is a correct translation of the compiler implementation from Section 4. Second, we have used Isabelle's built-in ML code generation feature [3,4] to compile the (verified parts of the) implementation from Section 4 using the functional compiling specification from Section 3 (see also [42]).

Of course, both approaches extend the trusted code base: the first by the translation validation tool, the second by Isabelle's code generation module. However, we can simply apply both methods and compare the resulting binaries. The probability that both produce the same error is negligible.

5 Conclusion and Future Work

We have sketched in this paper the correctness proof of a simple, non-optimizing $C0$ compiler. The correctness proof has been formalized in the theorem prover Isabelle / HOL and is split into a simulation theorem for the compiling specification and a proof for the total correctness (including termination and validity of guards) of the compiler implementation consisting of 1.500 lines of $C0$ code.[8] The formal proofs and definitions consist of roughly 85.000 lines of Isabelle code. This number covers the $C0$ small-step semantics (15.000 lines, including type correctness proofs and Theorem 2.1), the correctness proof for the compiler implementation (40.000 lines), and the simulation proof for the compiling specification (30.000 lines).

The compiler has been verified in the context of pervasive system verification in Verisoft. We had to deal with resource limitations on the target machine (e.g., restricted memory size) and other additional requirements; especially arguments about small-steps semantics have become mandatory. Thus, the top-level correctness theorem had to be extended with additional requirements on the $C0$ computations (e.g., limits on recursion depth), which guarantee that properties proved at the source language layer also hold for the compiled code.

We have also presented a solution to the boot strap problem. The compiling specification is in the executable subset of Isabelle's specification language. Thus, a trustworthy compiler binary can be generated by executing the specification.

The $C0$ implementation of a copying garbage collector has already been verified in Verisoft. However, the integration of this result into the compiler simulation theorem remains as future work.

[8] The formal proofs for the work presented in this paper can be downloaded from the Verisoft repository at http://www.verisoft.de/VerisoftRepository.html.

References

[1] Alkassar, E., M. Hillebrand, S. Knapp, R. Rusev and S. Tverdyshev, *Formal device and programming model for a serial interface*, in: B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, 2007, pp. 4–20.

[2] Alkassar, E., A. Starostin and N. Schirmer, *Formal pervasive verification of a paging mechanism*, in: *Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), to appear*, 2008.

[3] Berghofer, S., "Proofs, Programs and Executable Specifications in Higher Order Logic," Ph.D. thesis, Technical University of Munich (2003).

[4] Berghofer, S. and T. Nipkow, *Executing higher order logic*, in: *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science (LNCS) **2277** (2002), pp. 24–40.

[5] Beuster, G., N. Henrich and M. Wagner, *Real world verification – experiences from the verisoft email client*, in: G. Sutcliffe, R. Schmidt and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, 2006.

[6] Bevier, W. R., W. A. Hunt, Jr., J S. Moore and W. D. Young, *An approach to systems verification*, Journal of Automated Reasoning (JAR) **5** (1989), pp. 411–428.

[7] Beyer, S., C. Jacobi, D. Kroening, D. Leinenbach and W. Paul, *Putting it all together: Formal verification of the VAMP*, International Journal on Software Tools for Technology Transfer **8** (2006), pp. 411–430.

[8] Blazy, S., Z. Dargaye and X. Leroy, *Formal verification of a C compiler front-end*, in: J. Misra, T. Nipkow and E. Sekerinski, editors, *FM 2006: 14th International Symposium on Formal Methods*, LNCS **4085** (2006), pp. 460–475.

[9] Blech, J. O. and S. Glesner, *A formal correctness proof for code generation from SSA form in Isabelle / HOL.*, in: P. Dadam and M. Reichert, editors, *GI Jahrestagung (2)*, Lecture Notes in Informatics **51** (2004), pp. 449–458.

[10] Burstall, R., *Some techniques for proving correctness of programs which alter data structures*, in: B. Meltzer and D. Michie, editors, *Machine Intelligence 7* (1972), pp. 23–50.

[11] Chirica, L. M. and D. F. Martin, *Toward compiler implementation correctness proofs*, ACM Transactions on Programming Languages and Systems **8** (1986), pp. 185–214.

[12] Dalinger, I., "Formal Verification of a Processor with Memory Management Units," Ph.D. thesis, Saarland University, Computer Science Department (2006).

[13] Dave, M. A., *Compiler verification: A bibliography*, Association for Computing Machinery (ACM) SIGSOFT Software Engineering Notes **28** (2003).

[14] Dold, A. and V. Vialard, *A mechanically verified compiling specification for a Lisp compiler*, in: R. Hariharan, M. Mukund and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Sience*, LNCS **2245** (2001), pp. 144–155.

[15] Gargano, M., M. Hillebrand, D. Leinenbach and W. Paul, *On the correctness of operating system kernels*, in: J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS **3603** (2005), pp. 1–16.

[16] Goerigk, W. and U. Hoffmann, *Rigorous compiler implementation correctness: How to prove the real thing correct*, in: D. Hutter, W. Stephan, P. Traverso and M. Ullmann, editors, *Applied Formal Methods – FM-Trends 98*, LNCS **1641**, 1998, pp. 122–136.

[17] Gurevich, Y. and J. K. Huggins, *The semantics of the C programming language*, in: E. Börger, G. Jäger, H. K. Büning, S. Martini and M. M. Richter, editors, *CSL '92: Selected Papers from the Workshop on Computer Science Logic*, LNCS **702** (1993), pp. 274–308.

[18] Hillebrand, M., T. In der Rieden and W. Paul, *Dealing with I/O devices in the context of pervasive system verification*, in: *ICCD '05* (2005), pp. 309–316.

[19] In der Rieden, T. and A. Tsyban, *CVM - a verified framework for microkernel programmers*, in: *3rd intl Workshop on Systems Software Verification (SSV08)* (2008).

[20] "ISO 9899:1999: Programming languages – C," International Standardization Organization, 1999.

[21] Joyce, J. J., *Totally verified systems: Linking verified software to verified hardware.*, in: M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis*, LNCS **408** (1989), pp. 177–201.

[22] Klein, G. and T. Nipkow, *A machine-checked model for a Java-like language, virtual machine, and compiler*, ACM Transactions on Programming Languages and Systems **28** (2006), pp. 619–695.

[23] Leinenbach, D., "Compiler Verification in the Context of Pervasive System Verification," Ph.D. thesis, Saarland University, Computer Science Department, under appraisal (2008).

[24] Leinenbach, D., W. Paul and E. Petrova, *Towards the formal verification of a C0 compiler: Code generation and implementation correctness*, in: B. Aichernig and B. Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, 2005, pp. 2–11.

[25] Leroy, X., *Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant*, in: J. G. Morrisett and S. L. P. Jones, editors, *33rd symposium Principles of Programming Languages* (2006), pp. 42–54.

[26] Loeckx, J., K. Mehlhorn and R. Wilhelm, "Foundations of Programming Languages," John Wiley & Sons, Inc., 1989.

[27] McCarthy, J. and J. Painter, *Correctness of a compiler for arithmetic expressions*, in: J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, Proceedings of Symposia in Applied Mathematics **19** (1967), pp. 33–41.

[28] Mehta, F. and T. Nipkow, *Proving pointer programs in higher-order logic*, in: F. Baader, editor, *Conference on Automated Deduction (CADE)'03*, LNCS **2741** (2003), pp. 121–135.

[29] Moore, J S., *Piton: A verified assembly level language*, Technical Report 22, Comp. Logic Inc. Austin, Texas (1988).

[30] Moore, J S., *A grand challenge proposal for formal methods: A verified stack*, in: B. K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, LNCS **2757** (2003), pp. 161–172.

[31] The Motor Industry Research Association (MIRA), Ltd., UK, "MISRA-C:2004 — Guidelines for the use of the C language in critical systems," (2004).
URL http://www.misra-c2.com/

[32] Müller, S. M. and W. J. Paul, "Computer Architecture: Complexity and Correctness," Springer, 2000.

[33] Ni, Z., D. Yu and Z. Shao, *Using XCAP to certify realistic systems code: Machine context management*, in: K. Schneider and J. Brandt, editors, *TPHOLs*, LNCS **4732** (2007), pp. 189–206.

[34] Nielson, H. R. and F. Nielson, "Semantics with Applications: A Formal Introduction," John Wiley & Sons, Inc., New York, NY, USA, 1992, revised online version: 1999.

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

[36] Norrish, M., "C Formalised in HOL," Ph.D. thesis, University of Cambridge, Computer Laboratory (1998).

[37] Papaspyrou, N., "A Formal Semantics for the C Programming Language," Ph.D. thesis, National Technical University of Athens (1998).

[38] Petrova, E., "Verification of the C0 Compiler Implementation on the Source Code Level," Ph.D. thesis, Saarland University, Computer Science Department (2007).

[39] Pnueli, A., M. Siegel and E. Singerman, *Translation validation*, in: B. Steffen, editor, *TACAS '98*, LNCS **1384** (1998), pp. 151–166.

[40] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technical University of Munich (2006).

[41] Strecker, M., *Formal verification of a Java compiler in Isabelle*, in: A. Voronkov, editor, *CADE'02*, LNCS **2392**, pp. 63–77.

[42] Tzigarov, H., "Extended Oracle Proof Methods for Isabelle / HOL: Reflection and SMV Support," Diploma's thesis, Saarland University, Computer Science Department (2007).

[43] Zimmermann, W. and A. Dold, *A framework for modelling the semantics of expression evaluation with abstract state machines*, in: E. R. Egon Boerger, Angelo Gargantini, editor, *Abstract State Machines - Advances in Theory and Applications 10th International Workshop, ASM 2003*, LNCS **2589** (2003), pp. 391–406.

[44] Zimmermann, W. and T. Gaul, *On the construction of correct compiler back-ends: An ASM-approach*, Journal of Universal Computer Science **3** (1997), pp. 504–567.

Structured Types and Separation Logic

Harvey Tuch

Sydney Research Lab., National ICT Australia, Australia [1]
School of Computer Science and Engineering, UNSW, Sydney, Australia
harvey.tuch@nicta.com.au

Abstract

Structured types, such as C's arrays and `structs`, present additional challenges in pointer program verification. The conventional proof abstractions, multiple independent typed heaps and separation logic, which in previous work have been built on a low-level memory model for C and shown to be sound, are not directly applicable in verifications. This is due to the non-monotonic nature of pointer and lvalue validity in the presence of the unary &-operator. For example, type-safe updates through pointers to fields of a `struct` break the independence of updates across typed heaps or $\wedge^*$-conjuncts. In this paper we present a generalisation of our earlier formal memory model that captured the low-level features of C's pointers and memory and formed the basis for an expressive implementation of separation logic, with new features providing explicit support for C's structured types. We implement this framework in the theorem prover Isabelle/HOL and all proofs are machine checked.

Keywords: Separation Logic, C, Interactive Theorem Proving

1 Introduction

Programs featuring pointers are more difficult to verify than programs without indirection, largely as a result of the *aliasing* problem [1]. For example, consider a program with two pointer variables `float *` p and `int *` q and the following triple:

$$\{\!|\ \mathsf{True}\ |\!\}\ *p = 3.14;\ *q = 42;\ \{\!|\ *p = ?\ |\!\}$$

We are unable to ascertain the value pointed to by p as it may refer to the same location as q. With type-safe languages, this form of aliasing, which we call *inter-type aliasing*, can be ignored in proofs if the abstraction of multiple-typed heaps is used, where we have a semantic model with a heap function variable for each language type, e.g. $\mathsf{float\text{-}heap} :: \mathit{float\ ptr} \rightharpoonup \mathit{float}$, $\mathsf{int\text{-}heap} :: \mathit{int\ ptr} \rightharpoonup \mathit{int}$. Unfortunately in C we do not have this luxury as language features such as pointer arithmetic and casting break any illusion of type-safety, and we are forced to adopt the programmer's model of the heap as a function $\mathit{addr} \Rightarrow \mathit{byte}$, in particular when we wish to verify systems code exploiting compiler and architecture dependent language features.

[1] National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

A key observation is that while C permits code that violates memory and type safety, most code does remain within a type-safe fragment, and in earlier work we reconciled the *multiple-typed heaps* proof abstraction and this low-level view of memory, providing a rewriting approach to lifting proof states from byte granularity maps to typed heaps [13]. This avoided inter-type aliasing considerations where possible and gave a unified framework for proofs that needed to consider code that violated type-safety. The framework included a C parsing tool that emitted a mixed deep-shallow embedding in Schirmer's Hoare logic verification environment [11].

There still remains the problem of *intra-type aliasing* however, where pointers of the same type may alias one another. Again, it is possible to provide explicit conditions on states stating the presence or absence of aliasing, but this becomes rather cumbersome for inductively-defined data structures [1,10]. In particular, the *frame problem* limits the scalability of verifications. A potential solution is the *separation logic* of O'Hearn, Reynolds and others [5,10], providing a language for specifications and inference rules that both concisely allows for the expression of aliasing conditions in assertions and ensures modularity of specifications. In other previous work [14], we provided a shallow embedding of separation logic in Isabelle/HOL, building on the multiple-typed heaps development, resulting in a framework capable of accommodating different proof techniques to address aliasing.

In this paper, we extend this framework to further support C's structured types:

- We provide details of a deep embedding of structure type information capable of handling C's size, alignment, and padding restrictions as well as semantics for heap dereferencing for structured types.
- Earlier rewrites and proof rules for multiple-typed heaps and separation logic are generalised in such a way that they benefit from mechanisation and are still usable in verifications with little new overhead.
- Aspects of structured types that were previously handled in our semantics through shallow translation by trusted ML code are able to be promoted to the HOL level.

2 C `structs`

In our C-HOL type encoding, each C type was given a unique type in the theorem prover. All such types belonged to an axiomatic type class α::*c-type* in Isabelle, which introduced constants that connected the low-level byte representation and the HOL values:

$$\begin{array}{rr} \mathsf{to\text{-}bytes} :: \alpha{::}\textit{c-type} \Rightarrow \textit{byte list} & \mathsf{from\text{-}bytes} :: \textit{byte list} \rightharpoonup \alpha{::}\textit{c-type} \\ \mathsf{typ\text{-}tag} :: \alpha{::}\textit{c-type itself} \Rightarrow \textit{typ-tag} & \mathsf{typ\text{-}info} :: \alpha{::}\textit{c-type itself} \Rightarrow \textit{typ-info} \end{array}$$

The functions to-bytes and from-bytes converted between Isabelle values and lists of bytes suitable for writing to or reading from the raw heap state. The function typ-tag associated a unique *type tag* with each α::*c-type*, providing a means of treating language types as first-class values in HOL. Finally, typ-info allowed size and alignment information for the type to be calculated.

A distinct Isabelle pointer type for each Isabelle type, used to model C pointer types, was defined with:

$$\textbf{datatype}\ \alpha\ \textit{ptr} = \mathsf{Ptr}\ \textit{addr}$$

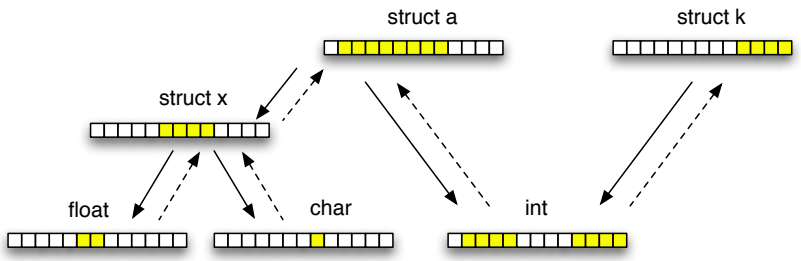

Fig. 1. Heap update dependencies.

The phantom α on the left-hand side was used to associate the pointer type information with pointer values in Isabelle's type system.

Primitive types such as `char` and `long *` could be defined in a library for each architecture/compiler in the expected way. `struct` types could be modelled at the HOL level with Isabelle **record** types. Trusted ML code in the C parser provided the following for each structured type used in a program:

- A corresponding **record** declaration
- Definitions of functions appearing in α::*c-type*, requiring full structure information to appear shallowly at the HOL level.
- Lvalue calculations, requiring the full structure information inside the ML parser, as well as offset/size/alignment calculations.

Example 2.1 As a running example, consider the following `struct` declarations:

```
struct x {      struct a {
  float y;        int b;
  char z;         struct x c;
};              };
```

The following triple demonstrates the most significant limitation with the earlier memory model:

$$\{\!|\ *p = (\!|\ y = 2.1,\ z = 'm'\ |\!)\ |\!\}\ p{\rightarrow}y = 1.2;\ \{\!|\ *p = ?\ |\!\}$$

The problem here is that even though the update and dereference are type-safe, and we do not need to consider aliasing, the proof rules we had developed so far considered this to be type-unsafe, as any region of memory could only have a single type, and p and $\&(p{\rightarrow}y)$ share a common address despite having different but related types. There is a similar problem for the effect of updates through `struct` references on enclosed field pointer values.

Fig. 1 demonstrates how this problem manifests itself in the multiple-typed heaps abstraction. It is no longer the case that updates to heaps can be treated independently, i.e.:

- Updating a field type's heap *may* affect typed heaps of enclosing `struct`s.
- Updating a `struct` affects typed heaps of field types (fields-of-fields, etc.).
- Update effects are no longer simple function update, they involve potentially multiple field updates and accesses.

The solution we propose in this paper is to treat structured type information as a first-class value in HOL and develop generalised definitions, rewrites and rules

making use of this. We treat structured types as first-class C types in the following to provide the benefits of abstraction and typing in proofs, even though at a semantic level they can be considered in terms of their members. Arrays in the heap decay to the corresponding pointer arithmetic, and inside structured values are also modelled using the definitions in §4.1. `union`s are treated differently, decaying to casts and *byte list*s as value representations.

3 Notation

Our meta-language Isabelle/HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types along with their primitive operations.

The space of total functions is denoted by $\Rightarrow$. Type variables are written α, β, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ. The option type

$$\textbf{datatype}\ \alpha\ \mathit{option} = \mathsf{None} \mid \mathsf{Some}\ \alpha$$

adjoins a new element $\bot$ to a type α. We use α *option* to model partial functions, writing $\lfloor a \rfloor$ instead of $\mathsf{Some}\ a$ and $\alpha \rightharpoonup \beta$ instead of $\alpha \Rightarrow \beta$ *option*. The Some constructor has an underspecified inverse called the, satisfying $\mathsf{the}\ \lfloor x \rfloor = x$. Function update is written $f(x := y)$ where $f :: \alpha \Rightarrow \beta$, $x :: \alpha$ and $y :: \beta$ and $f(x \mapsto y)$ stands for $f\ (x := \mathsf{Some}\ y)$. Domain restriction is $f\restriction_A$ where $f{::}\alpha \rightharpoonup \beta$ and $(f\restriction_A)\ x = (\mathsf{if}\ x \in A\ \mathsf{then}\ f\ x\ \mathsf{else}\ \bot)$.

Finite integers are represented by the type α *word* where α determines the word length. For succinctness, we use abbreviations like *word*8 and *word*32. The functions unat and $\mathsf{of\text{-}nat}$ convert to and from natural numbers (with *u* for *unsigned*).

Hoare triples are written $\{\!|P|\!\}\ c\ \{\!|Q|\!\}$ where P and Q are assertions and c a program. In assertions, we use the syntax $'x$ to refer to the program variable x in the current state, while ${}^{\sigma}x$ means x in state σ. Program states can be bound in assertions by $\{\!|\sigma.\ P|\!\}$.

Isabelle supports axiomatic type classes [16] similar to, but more restrictive than Haskell's. The notation $\alpha{::}\mathit{ring}$ restricts the type variable α to those types that support the axioms of class *ring*. Type classes can be reasoned about abstractly, with recourse just to the defining axioms. Further, a type τ can be shown to belong to a type class given a proof that the class's axioms hold in τ. All abstract consequences of the class's axioms then follow for τ.

For every Isabelle/HOL type α we can derive a type α *itself*, consisting of a single element denoted by $\mathsf{TYPE}(\alpha)$. This provides a convenient way to restrict the type of a term when working with polymorphic definitions.

4 Memory model

4.1 Type descriptions

The solution proposed in §2 requires that type meta-data be available at the HOL level. This needs to include information about the type structure, size, and alignment. In addition, a fine grained description of the value representation encoding

and decoding functions, such that it is possible to extract the functions for specific fields as well as the structure as a whole, is desirable.

At the HOL level, structure objects are represented using potentially nested Isabelle/HOL **record**s. Each field has access and update functions defined by the **record** package, e.g. for `struct a` represented as HOL record type *a-struct*, the functions b::*a-struct* ⇒ *int* and b-update::(*int* ⇒ *int*) ⇒ *a-struct* ⇒ *a-struct* are supplied — we write $v(\!|\mathsf{b} := x|\!)$ for b-update (K x) v. Where possible, it is helpful to use these **record** functions when reasoning about field accesses and updates, rather than the more detailed, lower-level view of fields as a subsequence of the byte-level value representation. To facilitate this, functions derived from the **record** functions are included in the type meta-data.

Definition 4.1 We can capture *abstract* record access and update functions for fields as *field descriptions*:

$$\begin{array}{lllll} \textbf{record} & \alpha\ \textit{field-desc} & = & \mathsf{field\text{-}access} :: \alpha \Rightarrow \textit{byte list} \Rightarrow \textit{byte list} \\ & & & \mathsf{field\text{-}update} :: \textit{byte list} \Rightarrow \alpha \Rightarrow \alpha \end{array}$$

These functions provide a connection between the structure's value as a typed HOL object and the value of a field in the structure as a *byte list*. field-access takes an additional *byte list* parameter, utilised in the semantics to provide the existing state of the *byte* sequence representing the field being described. This allows padding fields the ability to "pass through" the previous state during an update [2]. E.g. The field description for field b in `struct a` would be:

$$\begin{array}{l} (\!|\mathsf{field\text{-}access} = \mathsf{to\text{-}bytes} \circ \mathsf{b},\ \mathsf{field\text{-}update} = \lambda bs\ s.\ \mathsf{if}\ |bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\textit{int}) \\ \qquad \mathsf{then}\ s(\!|\mathsf{b} := \mathsf{from\text{-}bytes}\ bs|\!)\ \mathsf{else}\ s|\!) \end{array}$$

Definition 4.2 The type meta-data is captured in a *type description* with the following mutually-inductive definitions:

$$\begin{array}{lllll} \textbf{datatype} & \alpha\ \textit{typ-desc} & = & \mathsf{TypDesc}\ \alpha\ \textit{typ-struct}\ \textit{typ-name} \\ & \alpha\ \textit{typ-struct} & = & \mathsf{TypScalar}\ \textit{nat}\ \textit{nat}\ \alpha \\ & & & |\ \mathsf{TypAggregate}\ (\alpha\ \textit{typ-desc} \times \textit{field-name})\ \textit{list} \end{array}$$

A type description is a tree, with structures as internal nodes, branches labeled with field names and leaves corresponding to fields with primitive types. At leaves, size, alignment and an α is provided. A type description for `struct a` is given in Fig. 2.

There is not a one-to-one correspondence between fields in this structure and those in a C `struct`, as fields in this definition are also intended to explicitly represent the padding inserted by the compiler to ensure alignment restrictions are met. Type descriptions are specialised in two ways:

$$\begin{array}{lll} \alpha\ \textit{typ-info} & = & \alpha\ \textit{field-desc}\ \textit{typ-desc} \\ \textit{typ-uinfo} & = & (\textit{byte list} \Rightarrow \textit{byte list})\ \textit{typ-desc} \end{array}$$

The type information provides the information required to describe the encoding and decoding of the representation. Type information t can be "exported", with a function export-uinfo, to remove the α dependency with export-uinfo t, where leaf field descriptions are collapsed to *byte list* normalisation functions, i.e. an α *field-desc* d at a leaf with size n is replaced with λbs. field-access d (field-update d bs arbitrary) (replicate n 0).

[2] A more conservative, standard compliant approach, would be to use non-determinism or an oracle here.

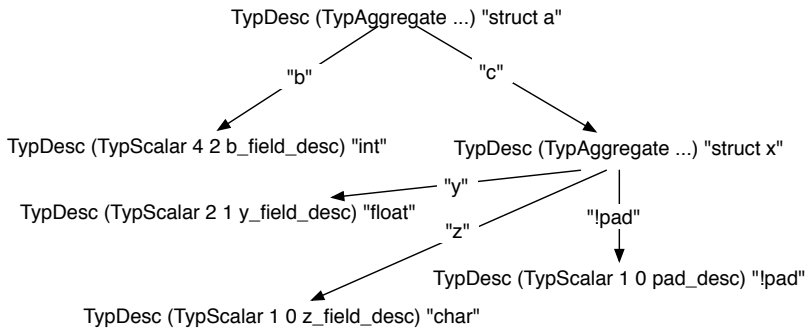

Fig. 2. Type description for `struct a`.

Normalisation is motivated by the observation that padding fields are ignored when reading structured values from their byte representation. Also, there may exist more than one byte representation for a value in C, even for primitive types. It provides us with a means to quantify over and compare C types.

The type information for a C type α is given by $\mathsf{TYPE}(\alpha)_\tau$ and we write $\mathsf{TYPE}(\alpha)_\nu$ for export-uinfo $\mathsf{TYPE}(\alpha)_\tau$.

Definition 4.3 A field name used to access and update structure fields with the C . and $\rightarrow$ operators can be viewed as a *field-name list* of .-separated fields leading to a sub-structure, which we refer to as a *qualified field name*. A qualified field name may lead to a field with a primitive or structure type, e.g. [] is the structure itself. Arrays members are named by index, e.g. $[''\text{-}\text{-}array\text{-}37'']$.

Table 1 provides a number of functions defined over type descriptions that we make use of in this paper. Here we summarise and provide examples — all functions are backed by primitive recursive definitions in Isabelle/HOL. $t \triangleright f$ performs "lookup", following a path f from the root of t and returning a sub-tree and offset if it exists. A related concept is td-set, where all sub-trees are returned. E.g.

$$\begin{array}{lcl}
\mathsf{TYPE}(a\text{-}struct)_\nu \triangleright [''c''] & = & \lfloor (\mathsf{TYPE}(x\text{-}struct)_\nu,\ 4) \rfloor \\
\mathsf{TYPE}(a\text{-}struct)_\nu \triangleright [''c'',\ ''b''] & = & \bot \\
\mathsf{td\text{-}set}\ \mathsf{TYPE}(x\text{-}struct)_\nu & = & \{(\mathsf{TYPE}(x\text{-}struct)_\nu,\ 0),\ (\mathsf{TYPE}(float)_\nu,\ 0),\ (\mathsf{TYPE}(char)_\nu,\ 2), \\
 & & (\mathsf{pad\text{-}export}\ 1,\ 3)\}
\end{array}$$

size-td and align-td are found by summing and taking the maximum of the leaf node sizes and alignments respectively. The latter is justified by the C standard's requirement that fields of aligned structures are themselves aligned. field-access-ti and field-update-ti compose their respective primitive leaf functions sequentially to provide the expected encoding and decoding functions for the aggregate type. E.g.

$$\begin{array}{lcl}
\mathsf{field\text{-}access\text{-}ti}\ \mathsf{TYPE}(a\text{-}struct)_\tau & = & \lambda v\ bs.\ \mathsf{to\text{-}bytes}\ (\mathsf{b}\ v)\ (\mathsf{take}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(int))\ bs)\ @\ \mathsf{to\text{-}bytes}\ (\mathsf{c}\ v) \\
 & & (\mathsf{take}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(x\text{-}struct))\ (\mathsf{drop}\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(int))\ bs))
\end{array}$$

Definition 4.4 The address corresponding to an *lvalue* designated by a structure field access or update can be found with:

$$\&(p{::}\alpha\ ptr \rightarrow f) \equiv \mathsf{ptr\text{-}val}\ p + \mathsf{of\text{-}nat}\ (\mathsf{snd}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_\nu \triangleright f)))$$

Lvalues appear in the semantics and proof obligations for statements like `p->f = v;`.

Definition 4.5 Finally, the connection between the HOL typed value, type information, size, alignment and underlying byte representation can be made through

- ▷ -	:: α *typ-desc* ⇒ *qualified-field-name* ⇀ α *typ-desc* × *nat* The sub-tree and offset from the base of the structure that a valid qualified field name leads to.
td-set	:: α *typ-desc* ⇒ (α *typ-desc* × *nat*) *set* The set of all sub-trees and their offset from the base of a structure.
size-td	:: α *typ-desc* ⇒ *nat* Type size, e.g. size-td TYPE(a-*struct*)$_\tau$ = 8.
align-td	:: α *typ-desc* ⇒ *nat* Type alignment exponent, e.g. align-td TYPE(a-*struct*)$_\tau$ = 2.
field-access-ti	:: α *typ-info* ⇒ (α ⇒ *byte list* ⇒ *byte list*) Derived field access for the entire structure represented by the type information.
field-update-ti	:: α *typ-info* ⇒ (*byte list* ⇒ α ⇒ α) Derived field update for the entire structure represented by the type information.
export-uinfo	:: α *typ-info* ⇒ *typ-uinfo* Export type information.

Table 1
Type description functions.

the following function definitions:

to-bytes (v::α) ≡ field-access-ti TYPE(α)$_\tau$ v from-bytes bs ≡ field-update-ti TYPE(α)$_\tau$ bs arbitrary
size-of TYPE(α) ≡ size-td TYPE(α)$_\tau$ align-of TYPE(α) ≡ 2 ^ align-td TYPE(α)$_\tau$

4.2 Type constraints

In this section we describe the fundamental properties that need to hold for each Isabelle/HOL type we use to model a C type. These ensure that the functions in Defn. 4.5 and the rest of §4.1 behave as expected by the C standard and in the proofs of the update rules. They are also available to the user of the framework.

Definition 4.6 The α::*mem-type* axiomatic type class requires the following size and alignment related properties to hold on a C type α for instantiation:

align-of TYPE(α) dvd size-of TYPE(α) size-of TYPE(α) < $|addr|$ align-of TYPE(α) dvd $|addr|$

These conditions follow mostly from requirements in the C standard, with the exception of the final alignment constraint which we add to make pointer arithmetic better behaved, and which holds on all the C implementations we are aware of. The constant $|addr|$ represents the size of the address space, e.g. 2^{32}.

The result of an entire structure update is independent of the original value:

$|bs|$ = size-of TYPE(α) ⟶ field-update-ti TYPE(α)$_\tau$ bs v = field-update-ti TYPE(α)$_\tau$ bs w

Three well-formedness conditions on the type information ensure sensible values for field names, node sizes and field descriptions:

wf-desc TYPE(α)$_\tau$ wf-size-desc TYPE(α)$_\tau$ wf-field-desc TYPE(α)$_\tau$

These conditions are now detailed in Defn. 4.7, Defn. 4.8 and Defn. 4.10.

Definition 4.7 We write wf-desc t when a type description t has no node with two or more branches labelled with the same field name.

Definition 4.8 We write wf-size-desc t when every node of the type description t has a non-zero size.

Definition 4.9 Type information t is *consistent* if the following properties hold:

$$\forall v\ bs\ bs'.\ |bs| = |bs'| \longrightarrow \mathsf{field\text{-}update\text{-}ti}\ t\ bs\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs'\ v) = \mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v$$
$$\forall v\ bs.\ |bs| = n \longrightarrow \mathsf{field\text{-}update\text{-}ti}\ t\ (\mathsf{field\text{-}access\text{-}ti}\ t\ v\ bs)\ v = v$$
$$\forall bs.\ |bs| = n \longrightarrow (\forall bs'.\ |bs'| = n \longrightarrow (\forall v\ v'.\ \mathsf{field\text{-}access\text{-}ti}\ t\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v)\ bs' = \mathsf{field\text{-}access\text{-}ti}\ t\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v')\ bs'))$$
$$\forall v\ bs.\ |bs| = n \longrightarrow |\mathsf{field\text{-}access\text{-}ti}\ t\ v\ bs| = n$$

where $n = \mathsf{size\text{-}td}\ t$. The properties are similar to those already provided by Isabelle's **record** package at the HOL level and can be established automatically.

Definition 4.10 Type information is well-formed w.r.t. field descriptions if all leaf fields are consistent, and for every pair of distinct leaf fields, s and t, the following properties hold:

$$\forall v\ bs\ bs'.\ \mathsf{field\text{-}update\text{-}ti}\ s\ bs\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs'\ v) = \mathsf{field\text{-}update\text{-}ti}\ t\ bs'\ (\mathsf{field\text{-}update\text{-}ti}\ s\ bs\ v)$$
$$\forall v\ bs\ bs'.\ |bs| = \mathsf{size\text{-}td}\ t \longrightarrow |bs'| = \mathsf{size\text{-}td}\ s \longrightarrow \mathsf{field\text{-}access\text{-}ti}\ s\ (\mathsf{field\text{-}update\text{-}ti}\ t\ bs\ v)\ bs' = \mathsf{field\text{-}access\text{-}ti}\ s\ v\ bs'$$

Again, these are standard commutativity and non-interference properties that we have at the HOL level and wish to preserve in field descriptions.

Theorem 4.11 *The* α*::mem-type axioms imply the following properties:*

$$\frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{\mathsf{from\text{-}bytes}\ (\mathsf{to\text{-}bytes}\ v\ bs) = v} \qquad \frac{|bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)}{|\mathsf{to\text{-}bytes}\ v\ bs| = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)} \qquad 0 < \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha)$$

4.3 Type combinators

The constraints of the previous section require both the construction of suitable type information and a corresponding α*::mem-type* instantiation proof for each type appearing in programs we wish to verify. This can be done entirely at the ML level, by synthesising both the intended HOL term for the type information directly, and a proof on the unfolded definition, but this is fragile and does not scale well.

An improved approach to type information construction is to do so using combinators that allow the structure to be built up field-wise and for which generic proof rules can be given. We use this approach and combinators and corresponding proof rules have been derived, but we elide for brevity.

4.4 Semantics

The C translation has a shallow HOL embedding as its target for expressions. Tuch et al [14] provide details of how side-effects and other aspects of the C semantics are translated, here we provide simply the definitions for the terms used to model heap accesses and updates.

Definition 4.12 Heap dereferences in expressions, e.g. $*p + 1$ are given a semantics by first lifting the raw heap state with the polymorphic lift function, e.g. lift $s\ p + 1$ where s is the current state.

$$\mathsf{heap\text{-}list} :: (addr \Rightarrow byte) \Rightarrow nat \Rightarrow addr \Rightarrow byte\ list$$
$$\mathsf{heap\text{-}list}\ h\ 0\ p \equiv []$$
$$\mathsf{heap\text{-}list}\ h\ (Suc\ n)\ p \equiv h\ p \cdot \mathsf{heap\text{-}list}\ h\ n\ (p + 1)$$

$$\mathsf{lift} :: (addr \Rightarrow byte) \Rightarrow \alpha :: c\text{-}type\ ptr \Rightarrow \alpha$$
$$\mathsf{lift}\ h \equiv \lambda p.\ \mathsf{from\text{-}bytes}\ (\mathsf{heap\text{-}list}\ h\ (\mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha))\ (\mathsf{ptr\text{-}val}\ p))$$

heap-update providing semantics for update dereferences:

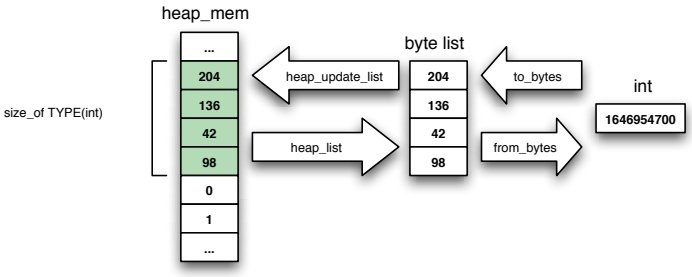

Fig. 3. *int* heap representation.

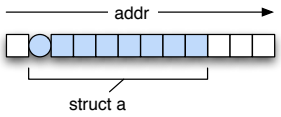

Fig. 4. Previous heap type description with a valid `struct a.` pointer

$$
\begin{array}{ll}
\textsf{heap-update-list} :: addr \Rightarrow byte\ list \Rightarrow (addr \Rightarrow byte) \Rightarrow (addr \Rightarrow byte) & \\
\textsf{heap-update-list}\ p\ []\ h & \equiv\ h \\
\textsf{heap-update-list}\ p\ (x{\cdot}xs)\ h & \equiv\ \textsf{heap-update-list}\ (p+1)\ xs\ (h(p := x)) \\
\textsf{heap-update}\ p\ (v{::}\alpha)\ h & \equiv\ \textsf{heap-update-list}\ (\textsf{ptr-val}\ p)\ (\textsf{to-bytes}\ v\ (\textsf{heap-list}\ h\ (\textsf{size-of TYPE}(\alpha))\ (\textsf{ptr-val}\ p)))\ h
\end{array}
$$

For example, $*p = *q + 5$ translates to the state transformer $\lambda s.$ heap-update p (lift $s\ q + 5$) s. Fig. 3 illustrates the above functions' value transformations.

4.5 Heap type description

Inside the type-safe fragment of C, where the majority of code remains, there is an implicit mapping between memory locations and types, and heap dereferences respect this mapping. In earlier work [14], we introduced this mapping as an additional state component, and referred to it as the *heap type description*:

$$heap\text{-}typ\text{-}desc = addr \rightharpoonup typ\text{-}tag\ option$$

The heap type description is a *history* variable, and as such does not influence the semantics of our programs. Since in C this mapping cannot be extracted from the source code, the program verifier adds proof annotations that update the heap type description. We wrote $d,g \models_t p$ to mean that the pointer p is valid in heap type description d with guard g. The guard g restricts the validity assertion based on the language's pointer dereferencing rules. This is depicted in Fig. 4.

The problem with this notion of the heap type description is that only a single pointer may be valid at any location. With structured types, we would like that at the base address a pointer for the structure type and that of the first field's type be valid. In general, for valid qualified field names f, we desire a validity monotonicity property, i.e. $d,g \models_t p \Longrightarrow d,g \models_t \textsf{Ptr}\ \&(p{\rightarrow}f)$.

To achieve this, we introduce a new definition for the heap type description:

$$
\begin{array}{lcl}
typ\text{-}slice & = & nat \rightharpoonup typ\text{-}uinfo \times bool \\
heap\text{-}typ\text{-}desc & = & addr \Rightarrow bool \times typ\text{-}slice
\end{array}
$$

Each location maps to a tuple, with the first component a *bool* indicating whether there is a value located at the address [3]. The second component is a *typ-slice*, providing an indexed map to the *typ-uinfo*s that may reside at a particular address. The index is calculated from the depth of the tree at an offset. The *bool* value indicates whether the location is the base or some other part of a value's footprint [4].

An example of the new heap type description is provided in Fig. 5. Each point is a *typ-uinfo* $\times$ *bool* pair, with the colour determined by the first component and shape by the second. Here an *a-struct* footprint extends on the horizontal axis above the footprints of its members. The vertical axis indicates a position in the *typ-slice* at the address. The second half of the *a-struct* is higher than the first, as the tree is deeper due to the *x-struct* changing the depth past this offset. An observation about the intuition behind pointer validity that can be taken from this figure is that it is independent of the presence or absence of type information from enclosing structured types in the history variable.

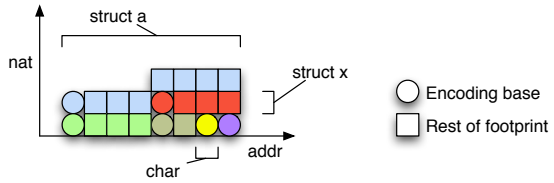

Fig. 5. New heap type description with a valid `struct a` pointer.

Definition 4.13 Pointer validity is defined for the heap type description as:

$$\begin{aligned}&\mathsf{valid\text{-}footprint}\ d\ x\ t \equiv \mathsf{let}\ n = \mathsf{size\text{-}td}\ t\\ &\qquad\qquad\mathsf{in}\ 0 < n \wedge\\ &\qquad\qquad\quad(\forall y{<}n.\ \mathsf{list\text{-}map}\ (\mathsf{typ\text{-}slice}\ t\ y) \subseteq_m \mathsf{snd}\ (d\ (x + \mathsf{of\text{-}nat}\ y)) \wedge\\ &\qquad\qquad\qquad\mathsf{fst}\ (d\ (x + \mathsf{of\text{-}nat}\ y)))\end{aligned}$$

$$d,g \models_t (p::\alpha\ ptr) \equiv \mathsf{valid\text{-}footprint}\ d\ (\mathsf{ptr\text{-}val}\ p)\ \mathsf{TYPE}(\alpha)_\nu \wedge g\ p$$

where $\mathsf{list\text{-}map}::\alpha\ list \Rightarrow (nat \rightharpoonup \alpha)$ converts a list to the expected map and typ-slice takes a vertical slice of the intended heap footprint from the exported type information at a given offset, e.g.:

$$\mathsf{typ\text{-}slice}\ \mathsf{TYPE}(a\text{-}struct)_\nu\ 4 = [(\mathsf{TYPE}(float)_\nu, \mathsf{True}), (\mathsf{TYPE}(x\text{-}struct)_\nu, \mathsf{True}), (\mathsf{TYPE}(a\text{-}struct)_\nu, \mathsf{False})]$$

The use of the map subset operator $\subseteq_m$ provides monotonicity.

As before, we have a retyping function ptr-retyp that updates the heap type description to make a given pointer valid. The definitions, properties and rules for this function are omitted for brevity.

5 Typed heaps

5.1 Lifting

The following two-stage lifting process provides an abstract heap view for proofs.

[3] This approach is taken in preference to a partial function to aid in partitioning state in §6.

[4] This is for same reason as in the previous approach to the heap type description, allowing consideration of the potential overlap of values of the same type to be eliminated for valid pointers.

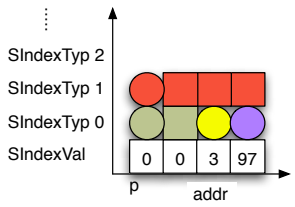

Fig. 6. Example *heap-state*.

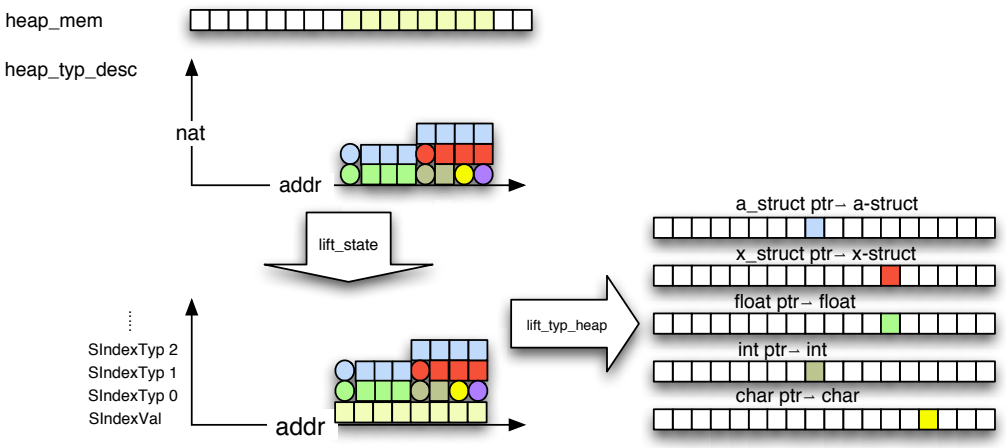

Fig. 7. Two-stage lifting.

Definition 5.1 The first stage, lift-state, results in an intermediate *heap-state*:

datatype *s-heap-index*	=	SIndexVal | SIndexTyp *nat*
datatype *s-heap-value*	=	SValue *byte* | STyp *typ-uinfo* × *bool*
s-addr	=	*addr* × *s-heap-index*
heap-state	=	*s-addr* ⇀ *s-heap-value*

An example of this state is provided in Fig. 6, with a *x-struct* footprint. The explanation for this model is provided in §6.1.

The function lift-state filters out locations that are False or ⊥ in the heap type description, depending on the index, removing values that should not affect the final lifted typed heaps. Equality between lifted heaps is then modulo the heap type description locations of interest for valid pointers.

```
lift-state ≡ λ(h, d) (x, y).
               case y of SIndexVal ⇒ if fst (d x) then ⌊SValue (h x)⌋ else ⊥
               | SIndexTyp n ⇒ option-case ⊥ (Some ∘ STyp) (snd (d x) n)
```

Lifted validity and heap-list are expressed on *heap-states* with $d,g \models_s p$ and heap-list-s respectively in the obvious way.

Definition 5.2 The second lifting stage results in typed lifted heaps again. The lift-typ-heap function restricts the heap domain so that the only locations affecting the resultant $\alpha\ ptr \rightharpoonup \alpha$ heap are valid pointer values. Equality is now modulo pointer validity.

```
lift-typ-heap g s ≡ (Some ∘ from-bytes ∘ heap-list-s s (size-of TYPE(α)) ∘ ptr-val)⌈{p | s,g ⊨s p}
```

The two stages, shown in Fig. 7, are combined with lift_τ:

$$\mathsf{lift}_\tau\ g \equiv \mathsf{lift\text{-}typ\text{-}heap}\ g \circ \mathsf{lift\text{-}state}$$

Like lift, lift_τ is polymorphic and returns an α typed heap. The program embedding continues to use the functions lift and heap-update, while pre/post conditions and invariants use the stronger lift_τ to make more precise statements.

5.2 Update dependency order

Definition 5.3 A partial order can be defined on type descriptions that expresses the update dependency between heaps, formalising the relation described in §2:

$$s \leq t \equiv \exists n.\ (s,\ n) \in \mathsf{td\text{-}set}\ t$$

This can be lifted to a predicate on α::*c-type itself* and β::*c-type itself*s:

$$s \leq_\tau t \equiv \mathsf{export\text{-}uinfo}\ \mathsf{TYPE}(\alpha)_\tau \leq \mathsf{export\text{-}uinfo}\ \mathsf{TYPE}(\beta)_\tau$$

Example 5.4 Using the running example, $\mathsf{TYPE}(\textit{x-struct}) <_\tau \mathsf{TYPE}(\textit{a-struct})$ and $\mathsf{TYPE}(\textit{int}) <_\tau \mathsf{TYPE}(\textit{a-struct})$. An update to an *a-struct* will always affect the lifted *int* heap, but an update of a *x-struct* will only sometimes affect the *a-struct* heap.

5.3 Rewrites

In this section we develop rewrites that allow the effects of updates on lifted typed heaps to be evaluated. First we present some auxiliary definitions and the key theorems, Thm. 5.7 and Thm. 5.9. These theorems have the form of conditional rewrites, but require some additional support to be efficiently applicable, so are followed by this detail.

Definition 5.5 A list of names of all fields matching an exported type information can be obtained with $\mathsf{field\text{-}names} :: \alpha\ \textit{typ-info} \Rightarrow \textit{typ-uinfo} \Rightarrow \textit{qualified-field-name list}$. E.g. $\mathsf{field\text{-}names}\ \mathsf{TYPE}(\textit{a-struct})_\tau\ \mathsf{TYPE}(\textit{float})_\nu = [[''c'',\ ''y'']]$.

Definition 5.6 From td-set, a predicate may be derived that checks whether a given pointer p::α *ptr* is to a field of a structured type with base q::β *ptr*:

$$\mathsf{field\text{-}of}\ p\ q \equiv (\mathsf{TYPE}(\alpha)_\nu,\ \mathsf{unat}\ (\mathsf{ptr\text{-}val}\ p\ -\ \mathsf{ptr\text{-}val}\ q)) \in \mathsf{td\text{-}set}\ \mathsf{TYPE}(\beta)_\nu\ 0$$

From $\triangleright$, functions may be derived that provide the first and second components of the result for a valid qualified field name:

$$\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\alpha)\ f \equiv \mathsf{fst}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_\tau \triangleright f)) \qquad \mathsf{field\text{-}offset}\ \mathsf{TYPE}(\alpha)\ f \equiv \mathsf{snd}\ (\mathsf{the}\ (\mathsf{TYPE}(\alpha)_\nu \triangleright f))$$

Theorem 5.7 *The lifted β heap following an update of a valid α ptr p, where α is a sub-type of β is given by:*

$$\frac{d,g' \models_t p \qquad \mathsf{TYPE}(\alpha) \leq_\tau \mathsf{TYPE}(\beta)}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{super\text{-}field\text{-}update}\ p\ v\ (\mathsf{lift}_\tau\ g\ (h,\ d))}$$

where

$$\begin{aligned}
&\mathsf{super\text{-}field\text{-}update}\ p\ v\ h \equiv \lambda q.\ \mathsf{if}\ \mathsf{field\text{-}of}\ p\ q \\
&\qquad \mathsf{then\ case}\ h\ q\ \mathsf{of}\ \bot \Rightarrow \bot \\
&\qquad\quad \mid \lfloor w \rfloor \Rightarrow \lfloor \mathsf{update\text{-}value}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\beta)_\tau\ \mathsf{TYPE}(\alpha)_\nu)\ v\ w \\
&\qquad\qquad\qquad (\mathsf{unat}\ (\mathsf{ptr\text{-}val}\ p\ -\ \mathsf{ptr\text{-}val}\ q)) \rfloor \\
&\qquad \mathsf{else}\ h\ q
\end{aligned}$$

Locations that do not enclose or are not valid β pointers are unaffected. The update is given by update-value:

$$\begin{array}{lcl}\mathsf{update\text{-}value}\ []\ (v{::}\alpha)\ (w{::}\beta)\ x & \equiv & w \\ \mathsf{update\text{-}value}\ (f{\cdot}fs)\ v\ w\ x & \equiv & \mathsf{if}\ x = \mathsf{field\text{-}offset}\ \mathsf{TYPE}(\beta)\ f\ \mathsf{then}\ \mathsf{field\text{-}update\text{-}ti}\ (\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\beta)\ f) \\ & & (\mathsf{to\text{-}bytes}_0\ v)\ w\ \mathsf{else}\ \mathsf{update\text{-}value}\ fs\ v\ w\ x\end{array}$$

This traverses the relevant fields of the enclosing structured type, looking for a field offset that matches the difference between the enclosing pointer base and p. If a match is found, update-value performs the update with the field's updator. We write $\mathsf{to\text{-}bytes}_0$ and $\mathsf{field\text{-}access\text{-}ti}_0$ when the supplied *byte list* is all zero.

While Thm. 5.7 gives a conditional rewrite that allows an update to be lifted to the typed heap level of §5.1, making use of the updated typed heap could involve unfolding this complex definition in general. However, additional rewrites can be given for well-behaved updates.

Theorem 5.8 *For a valid qualified field name, a* super-field-update *for a pointer* $\&(p{\rightarrow}f)$ *can be reduced to the field update obtained from the type information:*

$$\frac{\mathsf{TYPE}(\beta)_\tau \rhd f = \lfloor(s,\ n)\rfloor \qquad \mathsf{TYPE}(\alpha)_\nu = \mathsf{export\text{-}uinfo}\ s \qquad \mathsf{lift}_\tau\ g\ s\ p = \lfloor w\rfloor}{\mathsf{super\text{-}field\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}f))\ v\ (\mathsf{lift}_\tau\ g\ s) = \mathsf{lift}_\tau\ g\ s(p \mapsto \mathsf{field\text{-}update\text{-}ti}\ s\ (\mathsf{to\text{-}bytes}_0\ v)\ w)}$$

The $\rhd$ side-condition can be resolved without having to unfold the type information using rewrites installed during construction with combinators at the ML level. The field-update-ti is also rewritten to a **record** field updator. E.g.:

$$\mathsf{lift}_\tau\ g\ s\ p = \lfloor w\rfloor \Longrightarrow \mathsf{super\text{-}field\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}[''next''])\ v\ (\mathsf{lift}_\tau\ g\ s) = \mathsf{lift}_\tau\ g\ s(p \mapsto w(\!|\mathsf{next} := v|\!))$$

A rewrite can also be given for the two remaining cases, where $\mathsf{TYPE}(\beta) <_\tau \mathsf{TYPE}(\alpha)$ or $\mathsf{TYPE}(\alpha) \perp_\tau \mathsf{TYPE}(\beta)$.

Theorem 5.9 *The lifted* β *heap following an update of a valid* α *ptr* p*, where* α *is not a strict sub-type of* β *is given by:*

$$\frac{d{,}g' \models_t p \qquad \neg\ \mathsf{TYPE}(\alpha) <_\tau \mathsf{TYPE}(\beta)}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{sub\text{-}field\text{-}update}\ (\mathsf{field\text{-}names}\ \mathsf{TYPE}(\alpha)_\tau\ \mathsf{TYPE}(\beta)_\nu)\ p\ v\ (\mathsf{lift}_\tau\ g\ (h,\ d))}$$

where

$$\begin{array}{lcl}\mathsf{sub\text{-}field\text{-}update}\ []\ p\ (v{::}\alpha)\ s & \equiv & s{::}\beta\ ptr \rightharpoonup \beta \\ \mathsf{sub\text{-}field\text{-}update}\ (f{\cdot}fs)\ p\ v\ s & \equiv & (\mathsf{let}\ s' = \mathsf{sub\text{-}field\text{-}update}\ fs\ p\ v\ s\ \mathsf{in}\ s'(\mathsf{Ptr}\ \&(p{\rightarrow}f) \mapsto \mathsf{from\text{-}bytes} \\ & & (\mathsf{field\text{-}access\text{-}ti}_0\ (\mathsf{field\text{-}typ}\ \mathsf{TYPE}(\alpha)\ f)\ v)))\restriction_{\mathsf{dom}\ s}\end{array}$$

5.4 Non-interference

Theorem 5.10 *The rewrites for an update to a lifted typed heap through a valid pointer of the same type, or a disjoint type are the same as before [14]:*

$$\frac{d{,}g \models_t p}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{lift}_\tau\ g\ (h,\ d)(p \mapsto v)} \qquad \frac{d{,}g' \models_t p \qquad \mathsf{TYPE}(\alpha)_\nu \perp_t \mathsf{TYPE}(\beta)_\nu}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d) = \mathsf{lift}_\tau\ g\ (h,\ d)}$$

Bornat [1] describes multiple independent heaps based on distinct field names. Updates through a pointer dereference to a specific field only affect that heap. This does not work directly in the presence of the $\&(p{\rightarrow}f)$ operator and address arithmetic. However, the following can be shown:

Theorem 5.11 *When the base pointers are of the same type β, and neither of the field names are a prefix of the other, updates through an α pointer derived from one field do not affect the value in the γ lifted heap at the other:*

$$\frac{d,g' \models_t p \qquad d,ga \models_t q \qquad \mathsf{TYPE}(\beta)_\tau \triangleright f = \lfloor(s,\, m)\rfloor \qquad \mathsf{TYPE}(\beta)_\tau \triangleright f' = \lfloor(t,\, n)\rfloor \qquad \mathsf{size\text{-}td}\ s = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\alpha) \qquad \mathsf{size\text{-}td}\ t = \mathsf{size\text{-}of}\ \mathsf{TYPE}(\gamma) \qquad \neg\ f \leq f' \qquad \neg\ f' \leq f}{\mathsf{lift}_\tau\ g\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}f))\ v\ h,\ d)\ (\mathsf{Ptr}\ \&(q{\rightarrow}f')) = \mathsf{lift}_\tau\ g\ (h,\ d)\ (\mathsf{Ptr}\ \&(q{\rightarrow}f'))}$$

6 Separation logic

In this section we describe how the shallow embedding of separation logic [5,10] in Tuch et al [14] can be extended to structured types. The focus is on the singleton heap assertion $p \mapsto_g v$ as most of the other definitions and properties are standard.

6.1 Domain

We model separation assertions as predicates on *heap-state*s, applied in assertions of the verification environment to the result of the first lifting stage of §5.1. For example, a loop invariant with the separation assertion P and heap memory and type description state in the variables h and d respectively is written $\{\!|\ P\ (\mathsf{lift\text{-}state}\ (\acute{}h, \acute{}d))\ |\!\}$, which we abbreviate as $\{\!|\ P^{sep}\ |\!\}$.

The rationale for this choice of domain is that it allows for more expressive separation assertions than are possible with simpler models. From the earlier intermediate state, *addr* $\rightharpoonup$ *typ-tag option* $\times$ *byte* for unstructured types, a naive extension might be *addr* $\rightharpoonup$ *typ-uinfo list* $\times$ *byte*. Unfortunately, this does not allow for two assertions separated by $\wedge^*$ to refer to distinct type levels at the same address, necessary to provide flexible rules for retyping and unfolding, e.g. ignoring padding, we would expect that $(p \mapsto (\!|\ y = 3,\ z = 'r'\ |\!)) = (\mathsf{Ptr}\ (\&(p{\rightarrow}[''y'']))) \mapsto 3) \wedge^*$ $(\mathsf{Ptr}\ (\&(p{\rightarrow}[''z'']))) \mapsto 'r') \wedge^*$ typ-outline p, where typ-outline p contains the outer level type information for the enclosing structure. Adding a type level index to the domain of the *heap-state* provides this facility.

6.2 Shallow embedding

Definition 6.1 The s-footprint::α::*c-type ptr* $\Rightarrow$ *s-addr set* gives the set of addresses inside a pointer's *heap-state* footprint:

$$\begin{aligned}\mathsf{s\text{-}footprint}\ p \equiv \{&(\mathsf{ptr\text{-}val}\ p\ +\ \mathsf{of\text{-}nat}\ x,\\ &\ y)\ |\ x < \mathsf{size\text{-}td}\ \mathsf{TYPE}(\alpha)_\nu\ \wedge\\ &\quad (y = \mathsf{SIndexVal} \vee (\exists n.\ y = \mathsf{SIndexTyp}\ n \wedge n < |\mathsf{typ\text{-}slice}\ \mathsf{TYPE}(\alpha)_\nu\ x|))\}\end{aligned}$$

Definition 6.2 $p \mapsto_g v$ asserts that the heap contains exactly one mapping matching the guard g, at the location given by pointer p to value v:

$$p \mapsto_g v \equiv \lambda s.\ \mathsf{lift\text{-}typ\text{-}heap}\ g\ s\ p = \lfloor v\rfloor \wedge \mathsf{dom}\ s = \mathsf{s\text{-}footprint}\ p \wedge \mathsf{wf\text{-}heap\text{-}val}\ s$$

wf-heap-val asserts that the type, SValue or STyp, of a value in the *heap-state*, if present, matches the type of the index, SIndexVal or SIndexTyp respectively.

Definition 6.3 The standard definitions [10] for connectives can then be used, for the empty heap predicate, separation conjunction and implication these are:

$$
\begin{array}{lcl}
\square & \equiv & \lambda s.\ s = \mathsf{empty} \\
s_0 \perp s_1 & \equiv & \mathsf{dom}\ s_0 \cap \mathsf{dom}\ s_1 = \emptyset \\
s_0 +\!\!+ s_1 & \equiv & \lambda x.\ \mathsf{case}\ s_1\ x\ \mathsf{of}\ \perp \Rightarrow s_0\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor \\
P \wedge^* Q & \equiv & \lambda s.\ \exists s_0\ s_1.\ s_0 \perp s_1 \wedge s = s_1 +\!\!+ s_0 \wedge P\ s_0 \wedge Q\ s_1 \\
P \longrightarrow^* Q & \equiv & \lambda s.\ \forall s'.\ s \perp s' \wedge P\ s' \longrightarrow Q\ (s +\!\!+ s')
\end{array}
$$

Since this is a shallow embedding, standard HOL connectives and quantifiers can be freely mixed with the separation connectives, e.g. $\lambda s.\ P\ s \wedge (Q \wedge^* R)\ s$.

The standard commutative, associative, and distributive properties apply to the connectives, and we have formalised pure, intuitionistic, domain, and strictly exact assertions and properties [10]. The frame rule also still applies in this development.

6.3 Lifting proof obligations

Our verification condition generator applies weakest precondition rules to transform Hoare triples to HOL goals that can then be solved by applying theorem prover tactics. In §5.3, rewrites were given that could lift the raw heap component of these proof obligations, and in this section we provide rules that allow the low-level applications of lift and heap-update to be expressed in terms of separation assertions. This is desirable as reasoning can then use the derived rules for these assertions at the separation logic level.

Theorem 6.4 *The following rule connects* lift *and separation mapping assertions:*

$$\frac{(p \hookrightarrow_g v)\ (\mathsf{lift\text{-}state}\ (h,\ d))}{\mathsf{lift}\ h\ p = v}$$

Heap update dereferences produce proof goals of the form:

$$P\ (\mathsf{lift\text{-}state}\ (h,\ d)) \Longrightarrow Q\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p_0\ v_0\ (\mathsf{heap\text{-}update}\ p_1\ v_1\ (\mathsf{heap\text{-}update}\ p\ldots\ \ v\ldots\ (\mathsf{heap\text{-}update}\ p_n\ v_n\ h))),d))$$

Theorem 6.5 *To reduce* heap-update*s to the pre-state we can use:*

$$\frac{(p \mapsto_g w \wedge^* R)\ (\mathsf{lift\text{-}state}\ (h,\ d)) \qquad \mathsf{TYPE}(\beta)_\tau \triangleright f = \lfloor (s,\ n) \rfloor \qquad \mathsf{export\text{-}uinfo}\ s = \mathsf{TYPE}(\alpha)_\nu}{(p \mapsto_g \mathsf{field\text{-}update\text{-}ti}\ s\ (\mathsf{to\text{-}bytes}_0\ v)\ w \wedge^* R)\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(p{\rightarrow}f))\ v\ h,\ d))}$$

Thm. 6.5 can be applied in goals in similar situations to Thm. 5.8.

Theorem 6.6 *The earlier* heap-update *rules [14] still apply:*

$$\frac{(g \vdash_s p \wedge^* (p \mapsto_g v \longrightarrow^* P))\ (\mathsf{lift\text{-}state}\ (h,\ d))}{P\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d))} \qquad \frac{(g \vdash_s p \wedge^* R)\ (\mathsf{lift\text{-}state}\ (h,\ d))}{(p \mapsto_g v \wedge^* R)\ (\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ p\ v\ h,\ d))}$$

6.4 Unfolding

Additional rules can be given that allow one to dive inside a singleton heap assertion for a structured type value. This may be needed in extracting points-to information to aid in discharging guard proof obligations or side-conditions of some of the rules such as Thm. 6.4 and is useful in allowing the granularity of an assertion to be changed.

Theorem 6.7 *A points-to mapping assertion for a valid qualified field name can be derived from a singleton heap assertion with:*

$$\frac{(p \mapsto_g v)\ s \qquad \mathsf{TYPE}(\beta)_\tau \triangleright f = \lfloor (t,\ n) \rfloor \qquad \mathsf{export\text{-}uinfo}\ t = \mathsf{TYPE}(\alpha)_\nu \qquad \mathsf{guard\text{-}mono}\ g\ g}{(\mathsf{Ptr}\ \&(p{\rightarrow}f) \hookrightarrow_g \mathsf{from\text{-}bytes}\ (\mathsf{field\text{-}access\text{-}ti}_0\ t\ v))\ s}$$

```
struct node {
  int item;
  struct node *next;
};
```

```
struct node *reverse (struct node *ptr)
{
  struct node *last = NULL;

  while (ptr)
  {
    struct node *temp = ptr->next;

    ptr->next = last;
    last = ptr;
    ptr = temp;
  }
  return last;
}
```

Fig. 8. In-place list reversal C source code.

We have also developed a rewrite approach that unfolds fields for structured values — one can "zoom" in and out of structured values with this.

7 Example: In-place list reversal

Fig. 8 provides an example type-safe C program that performs in-place list reversal on a singly-linked list using a struct type to represent nodes.

This is a standard example in the separation logic and pointer program verification literature [10,7,14] and the pre/post specification and loop invariant are provided in Thm. 7.2.

Definition 7.1 The specification and invariant make reference to a list abstraction predicate, which lifts from a pointer-linked data structure in the heap to the corresponding algebraic data-type for a *node list* in Isabelle/HOL:

$$\begin{array}{ll} \mathsf{list}\ []\ i & \equiv\ \lambda s.\ i = \mathsf{NULL} \wedge \Box\ s \\ \mathsf{list}\ (x{\cdot}xs)\ i & \equiv\ \lambda s.\ i \neq \mathsf{NULL} \wedge (\exists j.\ \mathsf{item}\ j = x \wedge (i \mapsto_g j \wedge^*\ \mathsf{list}\ xs\ (\mathsf{next}\ j))\ s) \end{array}$$

Theorem 7.2 *The* `reverse` *function implements the following specification:*

$$\begin{array}{l} \forall\, zs.\ \{\!|(\mathsf{list}\ zs\ \acute{}ptr)^{sep}|\!\} \\ \quad \acute{}reverse\text{-}ret :== PROC\ reverse(\acute{}ptr) \\ \quad \{\!|(\mathsf{list}\ (\mathsf{rev}\ zs)\ \acute{}reverse\text{-}ret)^{sep}|\!\} \end{array}$$

Proof.

After running the verification condition generation, we are left with the 3 resulting proof obligations arising from the while Hoare logic rule, with the invariant:

$$\{\!|\exists\, xs\ ys.\ (\mathsf{list}\ xs\ \acute{}ptr \wedge^*\ \mathsf{list}\ ys\ \acute{}last)^{sep} \wedge \mathsf{rev}\ zs = \mathsf{rev}\ xs\ @\ ys|\!\}$$

The *Pre* $\Longrightarrow$ *Inv* and *Inv* $\Rightarrow$ *Post* conditions are trivial. Loop invariant preservation proof requires we show:

1. $\bigwedge zs\ a\ b\ last\ ptr\ ys\ list\ j.$
   $[\![ptr \neq \mathsf{NULL};\ \mathsf{rev}\ zs = \mathsf{rev}\ list\ @\ \mathsf{item}\ j{\cdot}ys;$
   $(ptr \mapsto_g j \wedge^*\ \mathsf{list}\ list\ (\mathsf{next}\ j) \wedge^*\ \mathsf{list}\ ys\ last)\ (\mathsf{lift\text{-}state}\ (a,\ b))]\!]$
   $\Longrightarrow (ptr \mapsto_g j(|\mathsf{next} := last|) \wedge^*$
   $\mathsf{list}\ ys\ last \wedge^*\ \mathsf{list}\ list\ (\mathsf{lift}\ a\ (\mathsf{Ptr}\ \&(ptr{\rightarrow}[''next''])))$
   $(\mathsf{lift\text{-}state}\ (\mathsf{heap\text{-}update}\ (\mathsf{Ptr}\ \&(ptr{\rightarrow}[''next'']))\ last\ a,\ b))$

This follows from Thm. 6.5. The first side-condition may be discharged with Thm. 6.4 and Thm. 6.7, eliminating the lift. The other side-conditions are dis-

charged by rewrites installed during C translation for evaluating $\triangleright$.

□

An interesting point in the proof is when we have to show:

1. $\bigwedge zs\ a\ b\ last\ ptr\ ys\ list\ j.$
   $[\![ptr \neq \mathsf{NULL};\ \mathsf{rev}\ zs = \mathsf{rev}\ list\ @\ \mathsf{item}\ j{\cdot}ys;$
   $(ptr \mapsto_g j \wedge^*\ \mathsf{list}\ list\ (\mathsf{next}\ j) \wedge^*\ \mathsf{list}\ ys\ last)\ (\mathsf{lift\text{-}state}\ (a,\ b))]\!]$
   $\Longrightarrow j(\!|\mathsf{next} := last|\!) = j(\!|\mathsf{next} := \mathsf{field\text{-}update\text{-}ti}\ \mathsf{TYPE}(node\ ptr)_\tau\ (\mathsf{to\text{-}bytes}_0\ last)\ (\mathsf{next}\ j)|\!)$

Here, applying the reverse definition of from-bytes and the α::*mem-type* axioms lifts the RHS to the HOL **record** level to simplify for the goal.

Compared to our earlier in-place list reversal example [14], the proof script was about the same structure and size, 67 lines. In our experience, lifts and heap-updates can be reduced as above for type-safe C, freeing the user from this level of detail. However, a completeness result is not possible in this shallow treatment.

8 Related work

The idea to use separate heaps for separate pointer types and structure fields in Hoare logic goes back to Burstall [2]. On the abstract level, our multiple typed heaps formalisation is most closely related to Bornat [1] and Mehta and Nipkow's [7] work in Isabelle, although we exploit Isabelle's type inference in a different way. We ground this abstract and efficient reasoning in a detailed C semantics that is directly applicable to concrete programs, and extend support to C's structured types. Moy [8] has also developed a memory model for C structured types and a type hierarchy. This differs from ours as it is based on physical sub-typing [12] and the focus of the work is on translating well-behaved unions and casts to sub-typing instances. The Caduceus tool [3] supports Hoare logic verification of C programs, including the type-safe part of pointer arithmetic at this level. We increase the applicability of program verification drastically by supporting the unsafe part as well. Separation logic [5,10] has been mechanised in theorem proving systems previously [15,6]. Again, we provide soundness for program verification by grounding these abstract, idealised models in a concrete semantics. We are able to support abstract separation logic notation and unsafe, low-level pointer manipulations at the same time.

On the semantics front Norrish [9] presents a very thorough and detailed memory model of C and our formalisation has similarities to exploratory work on C++ [4]. Our model unifies these low-level semantics with the proof abstractions of the previous paragraph.

9 Conclusion

In this paper we continued earlier work on pointer program verification in higher-order logic for C programs by providing extensions and generalisations resulting in a framework capable of fully exploiting C's structured types. We presented a development that deeply embeds type structure information in the theorem prover and generic rules to describe type-safe updates in two common interactive proof abstractions — multiple-typed heaps and separation logic. With the former, we

extended the earlier notion of heap independence to take into account a partial ordering of heap update dependency, and with the latter based the development on a heap state that allows for expressive assertions. Type-unsafe operations continue to be supported albeit at a proof cost.

Future work includes providing support for C's `union` types when they are well behaved, e.g. tagged unions, `struct` pointer casting in the case of physical subtyping, development of Isabelle tactics for separation logic proofs and integration with automated tools and decision procedures.

Acknowledgments

We thank Gerwin Klein for discussions and for reading drafts of this paper.

References

[1] Bornat, R., *Proving pointer programs in Hoare Logic*, in: R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, LNCS **1837** (2000), pp. 102–126.

[2] Burstall, R., *Some techniques for proving correctness of programs which alter data structures*, in: B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, Edinburgh University Press, 1972 pp. 23–50.

[3] Filliâtre, J.-C. and C. Marché, *Multi-prover verification of C programs*, in: *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, USA*, LNCS **3308** (2004), pp. 15–29.

[4] Hohmuth, M., H. Tews and S. G. Stephens, *Applying source-code verification to a microkernel — the VFiasco project*, Technical Report TUD-FI02-03-März, TU Dresden (2002).

[5] Ishtiaq, S. S. and P. W. O'Hearn, *BI as an assertion language for mutable data structures*, in: *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2001), pp. 14–26.

[6] Marti, N., R. Affeldt and A. Yonezawa, *Verification of the heap manager of an operating system using separation logic*, in: *Third workshop on Semantics, Program Analysis, and Computing Environments For Memory Management (SPACE 2006)*, 2006, pp. 61–72.

[7] Mehta, F. and T. Nipkow, *Proving pointer programs in higher-order logic*, Information and Computation (2005), to appear.

[8] Moy, Y., *Union and cast in deductive verification*, in: *C/C++ Verification Workshop*, Oxford, UK, 2007.

[9] Norrish, M., "C formalised in HOL," Ph.D. thesis, Computer Laboratory, University of Cambridge (1998).

[10] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *Proc. 17th IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[11] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technische Universität München (2006).

[12] Siff, M., S. Chandra, T. Ball, K. Kunchithapadam and T. Reps, *Coping with type casts in C*, in: *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering* (1999), pp. 180–198.

[13] Tuch, H. and G. Klein, *A unified memory model for pointers*, in: G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12)*, LNCS **3835**, 2005, pp. 474–488.

[14] Tuch, H., G. Klein and M. Norrish, *Types, bytes, and separation logic*, in: M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, 2007, p. 12.

[15] Weber, T., *Towards mechanized program verification with separation logic*, in: J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004*, Lecture Notes in Computer Science **3210** (2004), pp. 250–264.

[16] Wenzel, M., *Type classes and overloading in higher-order logic*, in: E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics'97*, LNCS **1275** (1997), pp. 307–322.

A Logic for Virtual Memory

Rafal Kolanski

Sydney Research Lab., National ICT Australia, Australia [1]
School of Computer Science and Engineering, UNSW, Sydney, Australia
rafal.kolanski@nicta.com.au

Abstract

We present an extension to classical separation logic which allows reasoning about virtual memory. Our logic is formalised in the Isabelle/HOL theorem prover in a manner allowing classical separation logic notation to be used at an abstract level. We demonstrate that in the common cases, such as user applications, our logic reduces to classical separation logic. At the same time we can express properties about page tables, direct physical memory access, virtual memory access, and shared memory in detail.

Keywords: Separation Logic, Virtual Memory, Interactive Theorem Proving

1 Introduction

Separation logic [14] has been used for verification of shared mutable data structures at the application level, such as those involved in C programs [18]. While effective, these techniques assume a view of memory as a function from addresses to values. For operating system verification, the situation is more complex. On hardware incorporating the virtual memory abstraction, two different virtual addresses may point to the same physical address. In the majority of operating system code this is not a problem, but the application view is insufficient for verifying the parts involved with the virtual mappings themselves, such as shared memory [16].

The virtual memory abstraction offers flexible, dynamic allocation of physical memory to running processes. It allows each process its own view of physical memory, called a virtual address space, via a set of virtual to physical address mappings. The mappings are stored in memory in a structure called the page table.

Applications usually deal only with their own data, but an operating system additionally manages the application's page table as well as its own. A memory write in this situation can result in the view of memory changing. Most of the time, it does not; in cases when it does, only *part* of the address space changes. Inferring

[1] National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

separation of what may have changed from what has not allows effective reasoning about such memory updates.

The virtual memory mechanism itself has been a target of verification [9], however apart from some work in progress [17] we have not encountered any work dealing with proofs about programs that make essential use of the virtual memory subsystem.

Our contribution in this work is a logic allowing effective reasoning about both the virtual and physical layers of memory. Additionally:

- The abstract layer of our separation logic is similar to traditional separation logic and in fact collapses to traditional separation logic for pure application reasoning, where the active page table is not mapped into virtual memory.
- Separating conjunction extends to multiply-mapped memory addresses.
- The low-level details of page table implementation are independent of the logic. We present an instantiation to a simple one-level page table to demonstrate the concept.
- All of the work presented in this paper is formalised in the Isabelle/HOL theorem prover [13].

Like [18], we use a shallow embedding of separation logic, meaning the constructs of assertions are translated to Isabelle/HOL, rather than being considered distinct types in the logic.

As this is a work in progress, we are using a simplified model: we use a simple machine abstraction and do not take into account memory permissions beyond whether a virtual address is mapped/unmapped. As such permissions are merely extra properties of virtual addresses, they can be easily integrated into our logic. Other work [18] has shown how to integrate actual machine encodings with separation logic. We believe our logic can be likewise connected.

While the normal way to present separation logics [14] is to describe a programming language, assertion language and resulting rules, we focus on the assertion language and properties holding on heap updates only. We do this because, like Tuch et al. [18], we aim to place our logic on top of a classical verification condition generator such as that of Schirmer [15]. The generator outputs higher-order logic statements involving the shallow embedding of our logic's assertions. Thus, the programming language is given, and the usual separation logic rules instead become rules about the semantic effect of the program on the heap.

2 Intuition

Separation logic is traditionally based on a model of memory as a partial function from addresses to values, called a heap. When reasoning about shared mutable data structures [14], such as the representation of memory in a language with pointers, separation logic offers a concise way of defining the disjointness of predicates in memory. The primary mechanism for doing this is the separating conjunction. As shown in Fig. 1, it works by dividing up the heap into two disjoint regions on which each side of the conjunction must hold respectively. For instance, it allows us to

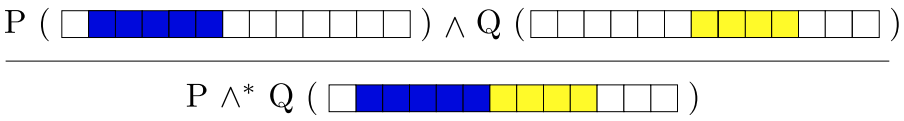

Fig. 1. Separation conjunction.

state two structures do not share memory.

As mentioned in the introduction, virtual memory is a hardware-enforced abstraction allowing each executing process its own view of memory, into which areas of physical memory can be dynamically inserted. It adds a further 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. Direct memory access can be approximated by using a one-to-one virtual-to-physical map.

To save space, virtual to physical mappings are stored in the page table with a granularity of pages whose size is usually dictated by the hardware. The page table resides in physical memory at a location called its root. The full set of virtual-to-physical mappings for a process can thus be lifted from physical memory if the root and type of page table are known.

Each access to a virtual address may result in a *page fault* in cases when the hardware fails to look up an address in the page table, for instance when that page resides on the disk, or has not yet been allocated. The page fault handler decides how to handle such cases. While our logic is aimed at dealing with situations useful in the verification of page fault handlers, we do not present page fault handler verification itself in this paper.

We will henceforth refer to physical memory as the *heap* in the spirit of traditional separation logic, and to the virtual-to-physical map as the *virtual map*.

Our logic is independent of the implementation of the page table. As an example, in this work we use a simple implementation: the one-level page table.

The addition of virtual memory to separation logic raises the issue of what exactly it means for two predicates to be separate, as well as what kind of state space we are to divide into sub-states in order to be able to express their disjointness. We want the ability to make statements on three levels: virtual to physical, physical to values and virtual to values. We also wish to preserve the usefulness of separating assertions in this context, as well as staying close to traditional separation logic notation.

Although the virtual map can be lifted from the heap via the page table, the heap alone is insufficient to make separation statements on all three levels. We observe this in the statement: under separating conjunction, virtual address vp maps to value x and the page table resides somewhere in physical memory. If we split the heap into page table and non-page table subheaps, there is no way to obtain vp's mapping from the non-page table subheap. It also requires carrying around the page table root as part of the state, causing divergence from traditional notation.

Instead, we propose a state consisting of *two maps*: the physical heap and the virtual map. Initially, we establish a valid state in which the virtual map is lifted out of the page table in the heap. Separation conjunction then splits *both* maps. Physical-to-value assertions work on the heap, virtual-to-physical on the virtual

map, and virtual-to-value on the composition of the two, which we will call the *address space*. This has two advantages:

- The page table and heap lifting is independent of our logic.
- Most operations on a single map make sense on a pair of maps, allowing us to use notation identical to standard separation logic and abstract away low-level details.

We begin with explaining basic notation in Sect. 3. In Sect. 4 we discuss the virtual memory abstraction and our specific instantiation of it. In Sect. 5 we introduce our logic, followed by examining its properties in Sect. 6. Finally, we discuss related work in Sect. 7 before concluding.

3 Notation

Our meta-language Isabelle/HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types along with their primitive operations.

The space of total functions is denoted by $\Rightarrow$. Type variables are written *'a*, *'b*, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ.

Pairs come with the two projection functions fst :: *'a* $\times$ *'b* $\Rightarrow$ *'a* and snd :: *'a* $\times$ *'b* $\Rightarrow$ *'b*. *Sets* (type *'a* set) follow the usual mathematical convention. Intervals are written as follows: $\{m..<n\}$ means $\{i \mid m \leq i < n\}$.

The option type

datatype *'a option* = None | Some *'a*

adjoins a new element None to a type *'a*. We use *'a option* to model partial functions, writing $\lfloor a \rfloor$ instead of Some a and *'a* $\rightharpoonup$ *'b* instead of *'a* $\Rightarrow$ *'b option*. The Some constructor has an underspecified inverse called the, satisfying the $\lfloor x \rfloor = x$. Function update is written $f(x := y)$ where f :: *'a* $\Rightarrow$ *'b*, x :: *'a* and y :: *'b* and $f(x \mapsto y)$ stands for f $(x :=$ Some $y)$. Domain restriction is $f \restriction_A$ where f::*'a* $\rightharpoonup$ *'b* and $(f \restriction_A)$ $x = ($if $x \in A$ then $f\ x$ else None$)$.

Finite integers are represented by the type *'a word* where *'a* determines the word length in bits.

Implication is denoted by $\Longrightarrow$ and $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$.

4 The Virtual Memory Environment

In this section, we describe the memory model our logic is currently based within: the pointer and page table abstractions, as well as their particular instantiations to our simple machine and one-level page table.

4.1 Pointer Abstraction and a Simple Machine

On all hardware platforms we are aware of, virtual and physical pointers are the length of the machine word, thus the same type. We use a generic pointer model

to prevent confusion between them, while still allowing functions working on both. This allows us to restrict parameters to a particular kind of pointer where necessary and overload functionality where not. We begin by wrapping each in a datatype

datatype *'a pptr-t* = PPtr *'a*
datatype *'a vptr-t* = VPtr *'a*

which we enclose in a type class *ptrs* encompassing both types. We define a value extraction function ptr-val satisfying

ptr-val (PPtr x) = x
ptr-val (VPtr x) = x

In order to disambiguate between pointer types in this work, we use *vp* for the virtual and *p* for physical pointers.

Using the above, we define an instantiation of a highly simplified machine: each 32-bit pointer points to a 32-bit word in memory for a total of 2^{32} words in an address space. Note that traditional implementations use 8-bit values. Our three machine types are: *vptr*, *pptr* and *val*, representing virtual pointers, physical pointers and values respectively. On this machine, we define the types of the three memory views described in Sect. 2 to be:

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

They represent the virtual map, heap and address space types respectively.

4.2 The Page Table Abstraction

We define our page table abstraction using Isabelle's modular reasoning construct called a *locale* [1]. It allows us to define the interface we require of a page table implementation independent of the hardware configuration: the functions each instantiation must provide as well as constraints they must satisfy.

The primary role of a page table is lookups in the virtual map it represents. In order to obtain this map, we require a lifting function ptable-lift. Given a heap and a page table root location, it obtains the virtual map.

ptable-lift::$('pptr \rightharpoonup 'val) \Rightarrow 'pptr \Rightarrow 'vptr \rightharpoonup 'pptr$

Not all page table roots are valid, due to the finiteness of the heap as well as other constraints such as alignment. valid-root defines what is and constitutes a valid location for the page table:

valid-root::$'pptr \Rightarrow bool$

Finally, in order to reason about when the page table is modified, we require information on the page table area, i.e. given a heap and a root pointer, which heap addresses constitute the table:

ptable-area::$('pptr \rightharpoonup 'val) \Rightarrow 'pptr \Rightarrow 'pptr$ set

Note that while the types in a locale are fixed throughout it, they are *abstract*. The *paddr*, *vaddr* and *val* we instantiated our simple machine to *can* instantiate *'paddr*, *'vaddr* and *'val* as their names suggest, but the page table interface specification is completely generic.

Definition 4.1 We require two properties on these functions. Firstly, the relation-

ship between lifting the page table to a virtual map and the page table area: the map must be lifted only out of the page table area. Secondly, that page table area must remain unchanged so long as heap updates don't touch it:

valid-root $r \Longrightarrow$ ptable-lift $h\ r =$ ptable-lift $(h\restriction_{\text{ptable-area}\ h\ r})\ r$
$[\![$valid-root $r;\ p \notin$ ptable-area $h\ r]\!] \Longrightarrow$ ptable-area $(h(p \mapsto v))\ r =$ ptable-area $h\ r$

Independent of the page table instantiation, theorem 4.2 follows.

Theorem 4.2 *Updating the heap outside the page table area has no effect on the resulting virtual map:*

$[\![$valid-root $r;\ p \notin$ ptable-area $h\ r]\!] \Longrightarrow$ ptable-lift $(h(p \mapsto v))\ r =$ ptable-lift $h\ r$

Definition 4.3 Using the supplied functions, we additionally define the concept of the page table not being mapped in the address space it defines:

ptable-not-mapped $h\ root \equiv$ let $vmap =$ ptable-lift $h\ root$ in $\forall\, v\ p.\ vmap\ v = \lfloor p \rfloor \longrightarrow p \notin$ ptable-area $h\ root$

As mentioned in Sect. 2, 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, its simplicity and contiguity allows for fast experimentation with particular memory layouts. 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:

lev1pt-pagesize	$\equiv$	2^{12}
lev1pt-size	$\equiv$	2^{32} div lev1pt-pagesize
get-page w	$\equiv$	ptr-val $w >>$ 12
ptr-remap $vp\ page$	$\equiv$	$page$ AND NOT 4095 OR vp AND 4095
ptable-lift $h\ r\ w$	$\equiv$	case $h\ r +$ get-page w of None $\Rightarrow$ None $\mid \lfloor addr \rfloor \Rightarrow$ if $addr$!! 0 then $\lfloor$PPtr (ptr-remap (ptr-val w) $addr$)$\rfloor$ else None
ptable-area $h\ r$	$\equiv$	$\{r..<r +$ lev1pt-size$\}$

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. This page table satisfies the properties required in Def. 4.1.

5 Extending Separation Logic

Having specified the nature of our memory environment, we will now describe our logic, its properties and relationship to classical separation logic as defined by Reynolds [14]. We will begin by comparing the state structure of the two logics, then follow with introducing all of the traditional separation logic constructs for our new setting, as well as a few constructs specific to our logic.

As mentioned in Sect. 2, our logic is based on a two-map state consisting of a heap and a virtual map. As a two-map is just a pair of maps, most map operations

still make sense. The operations on maps separation logic requires are map override, domain of, disjunction, subset, domain restriction and the empty map:

$$\begin{array}{lcl} m_1 \mathbin{+\!\!+} m_2 & \equiv & \lambda x.\ \mathsf{case}\ m_2\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow m_1\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor \\ \mathsf{dom}\ m & \equiv & \{a \mid m\ a \neq \mathsf{None}\} \\ m_1 \perp m_2 & \equiv & \mathsf{dom}\ m_1 \cap \mathsf{dom}\ m_2 = \emptyset \\ m_1 \subseteq_m m_2 & \equiv & \forall\, a \in \mathsf{dom}\ m_1.\ m_1\ a = m_2\ a \\ m\restriction_A & \equiv & \lambda x.\ \mathsf{if}\ x \in A\ \mathsf{then}\ m\ x\ \mathsf{else}\ \mathsf{None} \end{array}$$

Overriding ($++$) takes the value of an entry in the first map when it is not defined in the second. The domain of a map (dom) is the set of all defined values. Map disjointness ($\perp$) implies their domains are disjoint. A map is a subset of another ($\subseteq_m$) if all entries in the smaller map have the same values in the larger one. Finally, restricting a map ($\restriction$) is the same as restricting its domain.

We expand these to two-maps, using almost identical notation:

$$\begin{array}{lcl} (a,\ b) \mathbin{+\!\!+} (c,\ d) & \equiv & (a \mathbin{+\!\!+} c,\ b \mathbin{+\!\!+} d) \\ tdom\ (a,\ b) & \equiv & (\mathsf{dom}\ a,\ \mathsf{dom}\ b) \\ (a,\ b) \perp (c,\ d) & \equiv & a \perp c \wedge b \perp d \\ (a,\ b) \subseteq_t (c,\ d) & \equiv & a \subseteq_m c \wedge b \subseteq_m d \\ (a,\ b)\restriction_{(c,\ d)} & \equiv & (a\restriction_c,\ b\restriction_d) \\ \mathsf{empty} & \equiv & (\mathsf{empty},\ \mathsf{empty}) \end{array}$$

We will now introduce the standard separation logic constructs into our new setting.

We begin with the definition of separating conjunction, which our two-map abstraction allows us to express in an identical fashion to the traditional notation:

$$P \wedge^*\ Q \equiv \lambda s.\ \exists\, s_0\ s_1.\ s_0 \perp s_1 \wedge s = s_0 \mathbin{+\!\!+} s_1 \wedge P\ s_0 \wedge Q\ s_1$$

Note that s_0 and s_1 are now pairs of maps, not a heap as in traditional separation logic.

Separation logic also defines a useful concept for dealing with heap updates: separating implication. We say that P separately implies Q on a heap s when for any disjoint extension of s on which P holds, Q holds for s overridden with that extension. We can thus specify that modifying a heap in some way will establish a property. Once more, our version looks identical to the traditional one ($\longrightarrow$ is implication):

$$P \longrightarrow^*\ Q \equiv \lambda s.\ \forall\, s'.\ s \perp s' \wedge P\ s' \longrightarrow Q\ (s \mathbin{+\!\!+} s')$$

In order to make statements about the heap, traditional separation logic also provides several heap assertions: a concrete *maps-to*, as well as a *maps-to-something*:

$$p \mapsto v \equiv \lambda s.\ s\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ s = \{p\}$$

$$p \mapsto - \equiv \lambda s.\ \exists\, v.\ (p \mapsto v)\ s$$

$$p \hookrightarrow v \equiv p \mapsto v \wedge^*\ \mathsf{sep\text{-}true}$$

$$p \hookrightarrow - \equiv \lambda s.\ \exists\, v.\ (p \hookrightarrow v)\ s$$

The first two are domain exact [14], meaning they apply to a specific heap and are false for any extension of it. sep-true and sep-false are assertions defined to be respectively true and false for any heap:

$$\mathsf{sep\text{-}true} \equiv \lambda s.\ \mathsf{True}$$

$$\mathsf{sep\text{-}false} \equiv \lambda s.\ \mathsf{False}$$

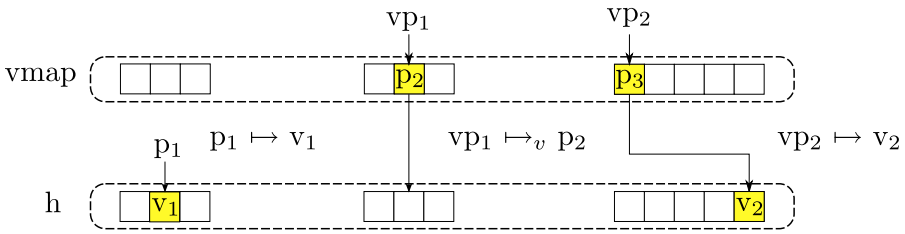

Fig. 2. Maps-to assertions on the heap, virtual map and address space.

The above assertions are standard separation logic assertions expressed on heaps. We will now proceed to describing their equivalents in our logic, expressed on a two-map state consisting of a heap and a virtual map.

Excepting the first domain exact assertion, which is the basis for all the others, we use identical notation to standard separation logic for heap and virtual map assertions, utilising Isabelle's function overloading ability.

In order to maintain the domain exact property of the first *maps-to* assertion, we proceed as follows. Heap assertions (leftmost on Fig. 2) are identical to normal separation logic, with the additional constraint that they only talk about the heap:

$$p \mapsto v \equiv \lambda(h,\ vmap).\ h\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ h = \{p\} \wedge vmap = \mathsf{empty}$$

Virtual map assertions (middle on Fig. 2) work similarly, but only on the virtual map. Unlike the other *maps-to* assertions, virtual addresses do not map to values. Hence, we use $\mapsto_v$ to denote them:

$$vp \mapsto_v p \equiv \lambda(h,\ vmap).\ vmap\ vp = \lfloor p \rfloor \wedge \mathsf{dom}\ vmap = \{vp\} \wedge h = \mathsf{empty}$$

Finally, address space assertions (rightmost on Fig. 2) involve both the heap and virtual map. In order to be domain exact, the *maps-to* assertion uses exactly one member of each, satisfying:

$$vp \mapsto v \equiv \lambda(h,\ vmap).\ \exists p.\ vmap\ vp = \lfloor p \rfloor \wedge h\ p = \lfloor v \rfloor \wedge \mathsf{dom}\ vmap = \{vp\} \wedge \mathsf{dom}\ h = \{p\}$$

With our new definitions, separating conjunction works as expected on predicates involving only one of the maps. However, address space predicates require entries from both maps. Thusly, under separating conjunction, if a virtual pointer $vp \mapsto v$ via some physical pointer p, then vp and p can not map to any other values than p and v, respectively:

$$\frac{(vp \mapsto v \wedge^* vp' \mapsto v')\ (h,\ vmap)}{vp \neq vp' \wedge vmap\ vp \neq vmap\ vp'} \qquad \frac{(vp \mapsto v \wedge^* p \mapsto v')\ (h,\ vmap)}{vmap\ vp \neq \lfloor p \rfloor}$$

In other words, mappings of virtual pointers to values will not share physical memory with each other, nor with mappings of physical pointers to values. Heap predicates and virtual map predicates are always disjoint, as they refer to two different maps.

Since virtual pointers can alias (map to the same physical address), we define an additional assertion to denote this case:

$$vp_1 \sim vp_2 \equiv \lambda(h,\ vmap).\ vmap\ vp_1 = vmap\ vp_2$$

We also found that being able to express that a set of pointers is mapped to some values, e.g. $\mathsf{ptable\text{-}area}\ r\ s \mapsto -$ proved very convenient. Again utilising overloading, we define it as:

$$\mathsf{pure}\ P \equiv \forall\, s\ s'.\ P\ s = P\ s'$$

$$[\![(P \wedge^* Q)\ s;\ \mathsf{pure}\ P;\ \mathsf{pure}\ Q]\!] \Longrightarrow P\ s \wedge Q\ s$$

$$[\![P\ s \wedge Q\ s;\ \mathsf{pure}\ P \vee \mathsf{pure}\ Q]\!] \Longrightarrow (P \wedge^* Q)\ s$$

$$\mathsf{pure}\ P \Longrightarrow (\lambda s.\ P\ s \wedge Q\ s) \wedge^* R = (\lambda s.\ P\ s \wedge (Q \wedge^* R)\ s)$$

$$[\![(P \longrightarrow^* Q)\ s;\ \mathsf{pure}\ P]\!] \Longrightarrow P\ s \longrightarrow Q\ s$$

$$[\![P\ s \longrightarrow Q\ s;\ \mathsf{pure}\ P;\ \mathsf{pure}\ Q]\!] \Longrightarrow (P \longrightarrow^* Q)\ s$$

Fig. 3. Pure assertions of our logic

$$\begin{aligned} S \mapsto - &\equiv \lambda s.\ \mathit{fold}\ \mathsf{op} \wedge^* (\lambda x.\ x \mapsto -)\ \square\ S\ s \\ S \hookrightarrow - &\equiv \lambda s.\ \forall p{\in}S.\ (p \hookrightarrow -)\ s \end{aligned}$$

The first of these simply states that all members of the set S map to some value, iteratively joined by $\wedge^*$.

The final traditional separation logic construct is empty heap assertion heap: $\square$. Our logic has three, depending on which of the maps we want to be empty:

$$\begin{aligned} \square_p &\equiv \lambda(h,\ \mathit{vmap}).\ h = \mathsf{empty} \\ \square_v &\equiv \lambda(h,\ \mathit{vmap}).\ \mathit{vmap} = \mathsf{empty} \\ \square &\equiv \square_p\ \lfloor\wedge\rfloor\ \square_v \\ P\ \lfloor\wedge\rfloor\ Q &\equiv \lambda x.\ P\ x \wedge Q\ x \end{aligned}$$

Where $\lfloor\wedge\rfloor$ is the lifted $\wedge$ operator.

We have now introduced adapted versions of all the separation logic connectives as well as those unique to our logic. This done, we can begin reasoning about its properties.

6 State Updates

In this section, we present various properties of our logic, describe how state updates work and their relationship to standard separation logic.

The basic mechanics of separation logic are intuitive about simple statements, as can be seen in these simple examples: no address may have two values; two different allocated physical addresses may have any value:

$$p \mapsto v_1 \wedge^* p \mapsto v_2 = \mathsf{sep\text{-}false}$$

$$[\![h = [p_1 \mapsto v_1,\ p_2 \mapsto v_2];\ p_1 \neq p_2]\!] \Longrightarrow (p_1 \mapsto v_1 \wedge^* p_2 \mapsto v_2)\ (h,\ \mathsf{empty})$$

We can lift these examples to the address space level, where two distinct virtual pointers can only be separated if they map to values via different physical addresses:

$$[\![s = ([p_1 \mapsto v_1,\ p_2 \mapsto v_2],\ [vp_1 \mapsto p_1,\ vp_2 \mapsto p_2]);\ vp_1 \neq vp_2;\ p_1 \neq p_2]\!] \\ \Longrightarrow (vp_1 \mapsto v_1 \wedge^* vp_2 \mapsto v_2)\ s$$

$$[\![s = ([p_1 \mapsto v_1,\ p_2 \mapsto v_2],\ [vp_1 \mapsto p_1,\ vp_2 \mapsto p_2]);\ vp_1 \neq vp_2;\ p_1 = p_2]\!] \\ \Longrightarrow \neg\ (vp_1 \mapsto v_1 \wedge^* vp_2 \mapsto v_2)\ s$$

Connectives in our logic conform to the associative, commutative and distributive properties of classical separation logic [14], with identical notation. We have also formalised the pure (Fig. 3) and intuitionistic (Fig. 4) assertions of separation logic (as they apply to our logic) and proved their properties [14], once more with no departure from standard notation.

intuitionistic $P \equiv \forall s\ s'.\ P\ s \wedge s \subseteq_t s' \longrightarrow P\ s'$

pure $P \Longrightarrow$ intuitionistic P

$[\![$intuitionistic P; intuitionistic $Q]\!] \Longrightarrow$ intuitionistic $(\lambda s.\ P\ s \wedge Q\ s)$

$[\![$intuitionistic P; intuitionistic $Q]\!] \Longrightarrow$ intuitionistic $(\lambda s.\ P\ s \vee Q\ s)$

$(\bigwedge x.$ intuitionistic $(P\ x)) \Longrightarrow$ intuitionistic $(\lambda s.\ \forall x.\ P\ x\ s)$

$(\bigwedge x.$ intuitionistic $(P\ x)) \Longrightarrow$ intuitionistic $(\lambda s.\ \exists x.\ P\ x\ s)$

intuitionistic (sep-true $\wedge^*$ P)

intuitionistic (sep-true $\longrightarrow^*$ P)

intuitionistic $P \Longrightarrow$ intuitionistic $(P \wedge^* Q)$

intuitionistic $Q \Longrightarrow$ intuitionistic $(P \longrightarrow^* Q)$

$[\![(P \wedge^*$ sep-true$)\ s;$ intuitionistic $P]\!] \Longrightarrow P\ s$

$[\![P\ s;$ intuitionistic $P]\!] \Longrightarrow$ (sep-true $\longrightarrow^* P)\ s$

Fig. 4. Intuitionistic assertions of our logic

The properties and simple examples above define a logic about heaps. In order to begin reasoning about programs, we need to consider their role as state transformers and hence examine the mechanics of heap updates and their effects.

In our machine instantiation, updating the heap alone is simply updating the map, though on more realistic instantiations it will become more complex:

heap-update $pptr\ val\ h \equiv h(pptr \mapsto val)$

In a virtual memory environment updating the heap can potentially involve changing the page table and thus the virtual map. Hence, when performing a state update at a physical address, we re-lift the page table from the updated heap:

heap-update-p $pptr\ val\ root \equiv \lambda(h,\ vmap).$ let $h' =$ heap-update $pptr\ val\ h$ in $(h',$ ptable-lift $h'\ root)$

As mentioned in Sect. 2, direct access to the heap is usually limited to system devices. Applications modify the heap through virtual addresses. The extra step involved over a direct heap update is the lookup of the virtual address. The page table is re-lifted as with heap-update-p:

heap-update-v $vp\ val\ root\ (h,\ vmap)\ \equiv$ case $vmap\ vp$ of None $\Rightarrow$ arbitrary
$\mid \lfloor p \rfloor \Rightarrow$ let $h' =$ heap-update $p\ val\ h$ in $(h',$ ptable-lift $h'\ root)$

Following Tuch et al. [18], heap updates on the physical layer always succeed. The behaviour of trying to access virtual addresses is more complex. At the hardware level, accessing an unmapped address will cause a *page fault* interrupt, which will redirect control to a page fault handler. The handler's implementation may then allocate pages, map pages, load them from disk, etc. If the fault is resolved successfully, control is returned to the faulting process and the access is attempted again. Hence, there are two cases: either the page is mapped or it is not. In the former, heap-update-v will perform the update just as the hardware would. In the latter, the term heap-update-v will not occur in the verification condition, as transferring control to the page fault handler will only change the necessary registers, not the heap. This means the arbitrary part of heap-update-v will not occur in verification conditions in either case.

At the application level, processes typically run in a simulated memory environ-

ment which abstracts away the inner workings of the memory subsystem. Hence, verification of application code under the assumption of page fault handler correctness can proceed by assuming that all memory implicitly required by the application (e.g. code, stack) is resident and mapped. Page fault handler correctness in this case is a simulation theorem showing that any execution with page faults can be simulated by one in the virtual application setting without faults.

At the operating system level, the virtual memory mechanism is visible. Furthermore, at least some system data structures, particularly in the kernel, need to be permanently resident in memory. To make this property part of the semantics, we can add guards to these critical kernel heap accesses, requiring the address to be mapped. For using the guard technique, see Tuch et al. [18].

Having defined heap updates, we now proceed to reasoning about their properties. Reynolds [14] defines two properties of state mutation, expressible in a shallow embedding (on a *one-heap* state) as:

$$\frac{(p \mapsto - \wedge^* P)\ h}{(p \mapsto v \wedge^* P)\ (\mathsf{heap\text{-}update}\ p\ v\ h)} \qquad \frac{(p \mapsto - \wedge^* (p \mapsto v \longrightarrow^* P))\ h}{P\ (\mathsf{heap\text{-}update}\ p\ v\ h)}$$

The former states that a property not dependent on the area of the heap being modified holds after the update. The latter is a weakest-precondition rule useful for backwards reasoning. It states that if P holds for a heap with the entry p set to v, then P will hold after the update, as that is precisely what heap-update does. Henceforth, we will refer to these as the *global* and *weakest-precondition* rules respectively.

As mentioned in Sect. 1, most operating system code falls into the "safe" category of using a one-to-one virtual map. Most application code likewise does not involve memory sharing or physical regions multiply mapped into virtual ones. In the following properties, we will present how our logic reduces down to statements identical to normal separation logic.

In all properties about updates in this section, we assume that r represents a valid root and s a state where the virtual map is the result of lifting the page table:

$$\mathsf{valid\text{-}root}\ r$$
$$\mathsf{ptable\text{-}lift}\ (\mathsf{fst}\ s)\ r = \mathsf{snd}\ s$$

Note that for physical addresses $p \mapsto -$ implies p is allocated, while for virtual addresses $vp \mapsto -$ implies that vp is mapped *and* the physical address it is mapped to is allocated.

The easiest to adapt is the physical update version of the global update rule.

Theorem 6.1 *For any physical pointer p that is not part of the page table, the classical global update rule holds:*

$$\frac{p \notin \mathsf{ptable\text{-}area}\ (\mathsf{fst}\ s)\ r \qquad (p \mapsto - \wedge^* P)\ s}{(p \mapsto v \wedge^* P)\ (\mathsf{heap\text{-}update\text{-}p}\ p\ v\ r\ s)}$$

As the page table is not modified, no mappings change, so the the virtual map lifted from the updated heap is identical to the original map. Apart from the requirement on p, our notation looks exactly like that of classical separation logic. We can rephrase the requirement on p in theorem 6.1 as a requirement on the page table area being allocated:

$$\frac{(p \mapsto - \wedge^* \ \mathsf{ptable\text{-}area}\ (\mathsf{fst}\ s)\ r \mapsto - \wedge^* \ P)\ s}{(p \mapsto v \wedge^* \ \mathsf{ptable\text{-}area}\ (\mathsf{fst}\ s)\ r \mapsto - \wedge^* \ P)\ (\mathsf{heap\text{-}update\text{-}p}\ p\ v\ r\ s)}$$

A similar transformation can also be done to the following update theorems, but we omit this form due to space considerations.

The weakest-precondition rule is more interesting, although it is derivable from the global update rule, due to its usefulness in backwards reasoning:

Theorem 6.2 *For any physical pointer p that is not part of the page table, the classical weakest-precondition update rule holds:*

$$\frac{p \notin \mathsf{ptable\text{-}area}\ (\mathsf{fst}\ s)\ r \qquad (p \mapsto - \wedge^* \ (p \mapsto v \longrightarrow^* \ P))\ s}{P\ (\mathsf{heap\text{-}update\text{-}p}\ p\ v\ r\ s)}$$

Once more the page table is not modified, meaning only one value in physical memory changes. If the old state overridden by the new value satisfies P, then P will hold after the update.

In the interest of brevity, we omit the simpler global update rule for virtual pointers, focusing on the weakest-precondition rule instead. The primary difference from theorem 6.2 is in the constraint of not residing in the page table being applied to the physical address the virtual pointer is mapped to:

Theorem 6.3 *For any virtual pointer vp, mapped to a physical address that is not part of the page table, the classical weakest-precondition update rule holds:*

$$\frac{\mathsf{the}\ (\mathsf{snd}\ s\ vp) \notin \mathsf{ptable\text{-}area}\ (\mathsf{fst}\ s)\ r \qquad (vp \mapsto - \wedge^* \ (vp \mapsto v \longrightarrow^* \ P))\ s}{P\ (\mathsf{heap\text{-}update\text{-}v}\ vp\ v\ r\ s)}$$

At the end of the operation, the address space is unchanged except for all the virtual pointers mapped to the physical address at which the heap was changed (via heap-update-v). As we described in Sect. 5 however, *vp* cannot alias with any other virtual pointer under separation conjunction, thus the modification of the heap will only be visible through one virtual address: *vp*.

Furthermore, if the page table is not mapped into the address space, as is typical for applications, heap-update-v cannot modify the page table at all:

Theorem 6.4 *The weakest-precondition rule holds for any virtual pointer vp if the page table is not mapped into the address space:*

$$\frac{\mathsf{ptable\text{-}not\text{-}mapped}\ (\mathsf{fst}\ s)\ r \qquad (vp \mapsto - \wedge^* \ (vp \mapsto v \longrightarrow^* \ P))\ s}{P\ (\mathsf{heap\text{-}update\text{-}v}\ vp\ v\ r\ s)}$$

As access to page tables is typically restricted to the operating system, this result demonstrates that for the application domain the update rules are identical to that of classical separation logic under the assumptions of a valid unchanging page table root and a valid state. For applications the page table root usually does not change, and the valid state condition can be checked automatically. Hence, for the application domain our logic reduces to classical separation logic.

Another observation is that applications cannot observe modifications to the heap at locations not mapped into their address space:

Theorem 6.5 *Under the assumption of the page table not being mapped into the current address space, for any physical address p not in the page table area and not mapped into the address space, updating the heap at p is not visible at the virtual memory level:*

$$\frac{p \notin \text{ptable-area } h\ r \qquad v = \text{ptable-lift } h\ r \qquad p \notin ran\ v \qquad \text{ptable-not-mapped } h\ r \qquad (new\text{-}h,\ new\text{-}vmap) = \text{heap-update-p } p\ val\ r\ (h,\ v)}{(new\text{-}vmap\ oo\ new\text{-}h)\ vp = (v\ oo\ h)\ vp}$$

where oo is the map composition operator:

$f\ oo\ g \equiv \lambda x.\ \text{case } f\ x \text{ of None} \Rightarrow \text{None} \mid \lfloor y \rfloor \Rightarrow g\ y$

The vast majority of code, both in applications and operating systems will fall into one of the above categories. However, the applicability of our logic does not end at situations where page tables are not touched. We identify three categories of page table updates, based on the modification of the virtual map: *map* (add entry), *remap* (modify entry) and *unmap* (remove entry). This is a work in progress: we have thus far formalised remap and are at present formalising map and unmap semantics. We will conclude this section with a discussion of remap.

As we are mostly interested in safety invariant preservation, we are concerned about what *does not* change during state updates. Even when updating the page table, most of the virtual map does not change.

Definition 6.6 The set of virtual addresses whose mappings are not affected during an update of the heap:

$\text{ptable-affected } f\ h\ root \equiv \text{let } vmap = \text{ptable-lift } h\ root \text{ in dom } vmap - vmap \cap_m \text{ptable-lift } (f\ h)\ root$

where $m_1 \cap_m m_2 \equiv \{x \in \text{dom } m_1.\ m_1\ x = m_2\ x\}$

During a remap operation, the domain of the virtual map remains the same, but its range might change. Thus, properties not invalidated by the heap modification and the change of virtual map must still hold.

Theorem 6.7 *Updating the heap via some virtual pointer vp preserves a property P if: the domain of the virtual map is not affected, vp's mapping is not affected and P is not invalidated by the new value at vp nor the new contents of the affected virtual map area.*

$$\frac{update = \text{heap-update-v } vp\ v\ r \qquad \text{dom } (\text{snd } (update\ s)) = \text{dom } (\text{ptable-lift } (\text{fst } s)\ r) \qquad (vp \mapsto - \wedge^* \text{ptable-affected } update\ r\ s \rightarrow_v - \wedge^* (vp \mapsto v \wedge^* \text{ptable-affected } update\ r\ s \rightarrow_v - \longrightarrow^* P))\ s}{P\ (\text{heap-update-v } vp\ v\ r\ s)}$$

7 Related work

The primary focus of this work is enhancement of separation logic, originally conceived by O'Hearn, Reynolds et al. [8,14]. Separation logic has previously been formalised in mechanised theorem proving systems [8,20]. We enhance these abstract models with the ability to reason about properties in a virtual memory environment.

Our exploration of virtual memory semantics is driven by the long-term goal of our research group: a verified operating system microkernel [5] based on L4. Earlier attempts such as UCLA Secure Linux [19], PSOS [11] and KIT [2] lacked the theorem proving technology required to deal with the complexities of a modern microkernel. Like our group, the VeriSoft project [6] is attempting to verify a

microkernel (VAMOS), but their focus is verifying an entire system stack, including compiler and applications. Our focus is on creating an efficient, verified microkernel.

Kernel verification efforts acknowledge the existence of virtual memory; previous work has involved verifying the virtual memory subsystem [9,4,7]. 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 [17]. 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. Hence, we believe our work to be a superset of the Robin approach.

Separation logic has been successfully applied to the verification of context switching code [10,12]. Tuch et al. demonstrated the extension of separation logic to reasoning about C programs involving pointer manipulation [18]. Presently, our work uses a simplified machine model with only one type and does not involve Hoare logic. We believe our framework supports addition of these extensions.

8 Discussion

Although our logic is similar to separation logic and collapses down to separation logic for pure application reasoning, it does not itself constitute a full separation logic. This is due to the fact that writes to the page table are not local actions [3]. In particular, the definitions of virtual-to-physical and virtual-to-value maps-to relations does not include the chunk of memory which contains the virtual mapping itself. As a result, we do not expect the frame rule [14] to hold for state updates involving page table modification.

The issue at the heart of this design decision is the granularity of virtual memory mappings, which causes attempts at pointing to the exact area of memory responsible for mappings to become problematic. For instance, on a 32-bit machine with a page size of 4096 bytes, a single-level page table and a page table entry size of four bytes, addresses 0 and 1 both receive their mappings from the same four-byte entry in the page table. To then say that address 0 maps to a value and *separately* address 1 maps to some value clearly causes a collision on those four bytes. Each entry thus maps 4096 addresses. This is one of the *simplest* setups; lifting this example to a *two-level* page table, the first-level entries map 1024 second-level entries, which in turn map 4096 addresses. Additionally, modern hardware commonly uses variable page sizes (superpages).

Our two-map method of reasoning about virtual memory thus sacrifices separation properties during page table modification in exchange for a simpler model, as well as easy abstraction over different hardware instantiations and multiple page table implementations.

The alternative to our two-map method would be to use a form of sub-byte

addressing, assigning multiple owners to slices of the bytes in each page-table entry. While potentially preserving all properties of separation logic, we believe this approach would make the memory model significantly more complex.

9 Conclusion and Future Work

We have presented an extension of separation logic which allows reasoning about virtual memory and processes running within it. Our logic allows for a convenient representation of predicates on memory at three levels: the virtual map, the physical heap and the virtual address space. The notation abstracts away details to the point of appearing very similar to classical separation logic. Our logic preserves the pure and intuitionistic properties of separation logic, again without exposing the underlying abstraction. Our work has been formalised in the Isabelle/HOL theorem prover.

We have shown that if the page table is not involved in an update or does not map itself, our logic reduces to normal separation logic.

Our work is highly modular. While we chose a simplified machine and page table implementation to aid with fast experimentation, the logic does not depend on the implementation of either. Although our framework does not presently have read/write access rights, it can be easily extended to encompass them. We aim to add this functionality in the near future.

As this is a work in progress, many applications and properties of our logic remain to be explored. The next step is more experimentation in the form of case studies on behaviours of programs in the presence of page table manipulation, possibly refining the model presented here into a complete separation logic as discussed in the previous section. Beyond that, we see the main direction for future work as extending our logic to handle C program verification in the style of Tuch, Klein, and Norrish [18].

Acknowledgements

We thank Gerwin Klein, Michael Norrish, Thomas Sewell and Harvey Tuch for suggestions, discussion and comments on earlier versions of this paper.

References

[1] Ballarin, C., *Locales and locale expressions in isabelle/isar*, in: S. Berardi, M. Coppo and F. Damiani, editors, *TYPES*, Lecture Notes in Computer Science **3085** (2003), pp. 34–50.

[2] Bevier, W. R., *Kit: A study in operating system verification*, IEEE Transactions on Software Engineering **15** (1989), pp. 1382–1396.

[3] Calcagno, C., P. W. O'Hearn and H. Yang, *Local action and abstract separation logic*, in: *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science* (2007), pp. 366–378.

[4] Dalinger, I., M. A. Hillebrand and W. J. Paul, *On the verification of memory management mechanisms*, in: D. Borrione and W. J. Paul, editors, *CHARME*, Lecture Notes in Computer Science **3725** (2005), pp. 301–316.

[5] Elphinstone, K., G. Klein, P. Derrin, T. Roscoe and G. Heiser, *Towards a practical, verified kernel*, in: *Proc. 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, 2007, p. 6, online proceedings at http://www.usenix.org/events/hotos07/tech/.

[6] Gargano, M., M. Hillebrand, D. Leinenbach and W. Paul, *On the correctness of operating system kernels*, in: *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, Oxford, UK, 2005, pp. 1–16.

[7] Hillebrand, M., "Address Spaces and Virtual Memory: Specification, Implementation, and Correctness," Ph.D. thesis, Saarland University, Saarbrcken (2005).
URL http://www-wjp.cs.uni-sb.de/publikationen/Hil05.pdf

[8] Ishtiaq, S. S. and P. W. O'Hearn, *BI as an assertion language for mutable data structures*, in: *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2001), pp. 14–26.

[9] Klein, G. and H. Tuch, *Towards verified virtual memory in L4*, in: K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[10] Myreen, M. O. and M. J. C. Gordon, *Hoare logic for realistically modelled machine code*, in: O. Grumberg and M. Huth, editors, *TACAS*, Lecture Notes in Computer Science **4424** (2007), pp. 568–582.

[11] Neumann, P. G., R. S. Boyer, R. J. Feiertag, K. N. Levitt and L. Robinson, *A provably secure operating system: The system, its applications, and proofs*, Technical Report CSL-116, SRI International (1980).

[12] Ni, Z., D. Yu and Z. Shao, *Using xcap to certify realistic systems code: Machine context management*, in: K. Schneider and J. Brandt, editors, *TPHOLs*, Lecture Notes in Computer Science **4732** (2007), pp. 189–206.

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

[14] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *Proc. 17th IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[15] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technische Universität München (2006).

[16] Tews, H., *Well-behaved memory on top of virtual memory*, Presentation at the NICTA International Workshop on System Verification, Sydney (2006).

[17] Tews, H., *Formal methods in the robin project: Specification and verification of the nova microhypervisor*, Submitted to the C/C++ Verification Workshop (2007), available from www.cs.ru.nl/~tews/science.html.

[18] Tuch, H., G. Klein and M. Norrish, *Types, bytes, and separation logic*, in: M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, 2007, pp. 97–108.

[19] Walker, B., R. Kemmerer and G. Popek, *Specification and verification of the UCLA Unix security kernel*, CACM **23** (1980), pp. 118–131.

[20] Weber, T., *Towards mechanized program verification with separation logic*, in: J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004*, Lecture Notes in Computer Science **3210** (2004), pp. 250–264.

A Formal Model of Memory Peculiarities for the Verification of Low-Level Operating-System Code

Hendrik Tews [1, 2] Tjark Weber [1, 3]

Institute for Computing and Information Sciences
Radboud Universiteit Nijmegen, The Netherlands

Marcus Völp [1, 4]

Institute for System Architecture
Technische Universität Dresden, Germany

Abstract

This paper presents our solutions to some problems we encountered in an ongoing attempt to verify the micro-hypervisor currently developed within the Robin project. The problems that we discuss are (1) efficient automatic reasoning for type-correct programs in virtual memory, and (2) modeling memory-mapped devices with alignment requirements. The discussed solutions are integrated in our verification environment for operating-system kernels in the interactive theorem prover PVS. This verification environment will ultimately be used for the verification of the Robin micro-hypervisor. As a proof of concept we include an example verification of a very simple piece of code in our environment.

Keywords: operating-system kernel, micro-hypervisor, virtual memory, memory-mapped devices, formal verification

1 Introduction

The programming environment of operating-system kernels differs in essential ways from that of application programs. The most prominent differences are direct hardware access and privileged processor instructions. In addition, certain situations that are absurd from an application-programming point of view are possible (and sometimes even very common) in kernel programming. For instance, many kernels see some piece of main memory at different virtual addresses. (Other important

[1] The authors have been supported by the European Union through PASR grant 104600.

[2] Homepage: http://www.cs.ru.nl/~tews

[3] Homepage: http://www.cs.ru.nl/~weber

[4] Homepage: http://os.inf.tu-dresden.de/~voelp

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

ways in which kernel programming differs are the use of casts and pointer arithmetic, however, those are outside the main scope of this paper.)

The additional hardware features that are exploited in a kernel programming environment are usually subject to very specific programming rules, which are described in the hardware architecture's technical documentation. Typically the rules are not enforced and, when not obeyed, one might get bugs that are hard to reproduce. As a consequence certain kinds of bugs can only occur in kernels (or similar kinds of low-level systems).

In the following we use the term *kernel-programming features* to refer to the additional phenomena of kernel programming just described. A verification environment for kernel code must of course model the kernel-programming features in order to give a semantics to the code. Less immediate but equally important is, however, that the verification environment is faithful with respect to the possible bugs associated with kernel-programming features. With faithful we mean here that any such bug must with certainty lead to a verification failure. The subtle differences between valid and erroneous code make the design of the verification environment very challenging.

In this paper we discuss our approach to model certain kernel-programming features that are used in the Robin micro-hypervisor. The solutions presented here are already implemented in our verification environment. The presentation includes a discussion of the kinds of programming errors that our models are able and are not able to catch. We focus on the memory peculiarities in this paper and present the following two points:

- A model of virtual memory that captures virtual-memory aliases, permits the verification of page-table modifications and efficient reasoning about well-behaved code (Section 3).
- A general model for memory-mapped devices and the phenomena of reserved bits in certain hardware registers (Section 4).

In addition, Section 2 gives a general overview of our verification environment, and Section 5 contains a verification example.

2 Overview of the Robin Verification Environment

This section provides some details about our verification environment for the Robin micro-hypervisor, see Figure 1 for illustration. More technical information can be obtained from [8], while the context of the Robin project is described in [6]. Our approach relies on source-code verification in the interactive theorem prover PVS [5]. The input language of PVS is higher-order logic enriched with predicate subtyping and some other forms of dependent types. For the verification we model parts of the IA32 hardware and the semantics of C++ data types inside PVS. These two models provide the basic operations for a model of the micro-hypervisor. Then we use the prover

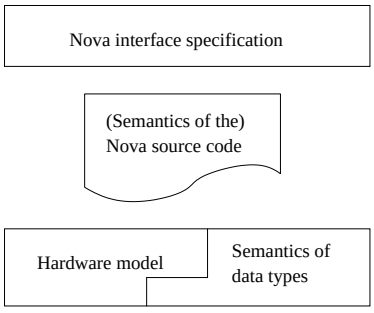

Fig. 1: Verification approach

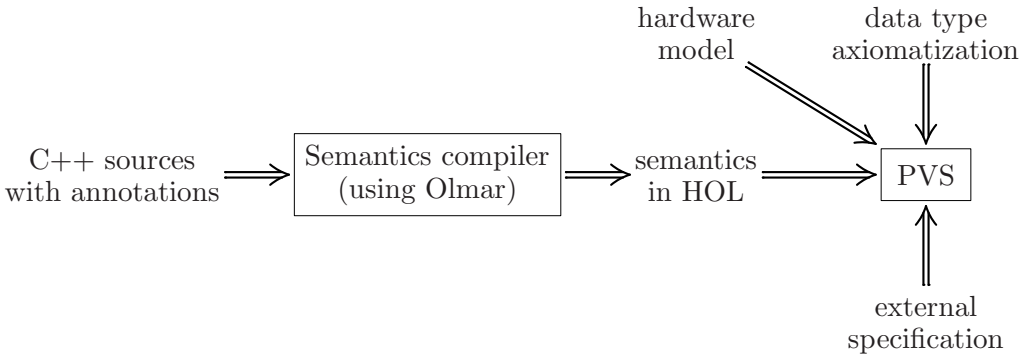

Fig. 2. Approach for source code verification

component of PVS to establish theorems about the model. Technically we show

$$\Phi_{data_types},\ \Phi_{hardware} \vdash \varphi(\text{hypervisor}),$$

where φ is one property from the hypervisor specification, such as *termination without runtime type errors*. Our verification results will describe properties of the source code. A formal lifting of the results to object code (which would eliminate the correctness of the compiler from our assumptions) is planned for the future.

Figure 2 depicts the data flow of our verification approach. A semantics compiler translates the C++ source code into its semantics in higher-order logic in PVS. The semantics compiler is currently developed on the basis of the Elsa/Olmar C++ front-end [7] and will be described elsewhere. The basic building blocks of the C++ semantics are provided by the hardware model and the data-type semantics.

The operations in the hardware model as well as the data-type semantics and the semantics of C++ fragments are uniformly modeled as *state transformers*. State transformers come in two flavors: statement state transformers (for C++ statements) and expression state transformers (for C++ expressions and everything else). An expression state transformer is a function of type

$$\mathsf{State} \to \mathsf{ExprResult[State, Data]}.$$

Here State is the type of all possible states of the hardware model, and Data is a type parameter for the result of the state transformer (if it terminates successfully). Both types are theory parameters in our PVS formalization, making them effectively polymorphic. In the verification of concrete C++ programs however, Data is instantiated with a fixed type for each state transformer. The State parameter is either instantiated as well (if we verify against a concrete hardware model), or left polymorphic if we verify against the plain-memory specification (see Section 3) in general. The type ExprResult is defined as follows:

```
ExprResult[State, Data : Type] : Datatype
BEGIN
  OK(state: State, data: Data) : OK?
  Exception(ex_type : Exception_type, state : State) : Exception?
  Fatal : Fatal?
  Hang : Hang?
END ExprResult
```

This piece of PVS code defines ExprResult as a disjoint union with four variants tagged OK, Exception, Fatal and Hang. The identifiers with question marks are recognizer predicates for the corresponding variants (e.g., OK? is true on OK(···) and false on the other three variants). The identifiers state, data, ex_type are (partial) accessor functions (e.g., state(OK(s, −)) = s).

A state-transformer result of the form OK(s, d) models successful termination with successor state s and result d. Hang stands for non-termination, for instance because of a while loop or a page fault that keeps occurring at the same instruction. Fatal is reserved for unrecoverable errors, which we want to rule out by verification. A result of the form Exception models hardware exceptions and interrupts that will be handled by the micro-kernel. [5]

Statement state transformers have the form [State → StmtResult[State, Data]] with a very similar type StmtResult. The main difference between ExprResult and StmtResult is that StmtResult contains abnormalities like Break and Return to model the corresponding C++ control-flow statements, very similar to [3]. Further, OK does not carry a data element. The parameter of type Data is used inside Return to model the return type of C++ functions.

State transformers can be composed in the obvious way. For two state transformers f and g their composition f ## g is a state transformer that performs the effect of g on the successor state of f if f returns OK. Otherwise g is discarded and the result of f is the result of the composition. If f is an expression transformer, the data in any OK result is discarded.

The first base component of our verification environment, the hardware model, formalizes an abstract model of the IA32 hardware in PVS. It provides physical memory, virtual memory with address translation via page tables, and much more. The hardware model does not blindly implement the behavior of the real hardware. Instead certain subtle programming errors that would cause the real CPU to do nonsense yield unprovable proof obligations in the hardware model. For instance, the attempt to interpret a string as a page-table entry yields a proof obligation that remains unprovable unless a suitable data-type conversion is formalized as an axiom. (Of course such an axiom is not justified.) This kind of error checking even works for hardware-initiated page-table traversals during address translation.

The memory formalization in the hardware model is split into different layers of memory models, for instance for physical and virtual memory. These different memory models share a common interface. Every memory model defines a type State of possible states and the following record of operations.

```
Memory_struct : Type = [#
    memory_read : [Address → [State → ExprResult[State, Byte]]],
    memory_write : [Address, Byte → [State → ExprResult[State, Unit]]],
    ... #]
```

The two operations respectively read and write one byte at the given address. Note that reading in memory can change the memory (for instance set the accessed bits in the page table). The memory structure contains two additional operations for

[5] Exception does *not* model C++ exceptions. We consider C++ exceptions too heavyweight to be used in an operating-system kernel. Therefore they are outside the fragment of C++ treated in this paper.

modeling memory-mapped devices and reserved bits, see Section 4.

In order to get a uniform treatment of memory-mapped devices and the special effects of feature flags in CPU control registers, we decided to have a uniform address space for both memory and registers. The type Address is therefore defined as a record consisting of a Register_Id and an offset:

Address : **Type** = [# type_of : Register_Id, offset : nat #]

Real memory appears as a (rather big) special register with Register_Id Mem. For memory the offset is the real address. For hardware registers the offset will most often be 0, however, for processor architectures featuring partial register access, the offset might be positive (e.g., for accessing `AH` one would use offset 1 in `EAX`). The sizes and possible offsets of hardware registers are enforced with suitable side effects, see Section 4.

On top of the memory structure we define the following two functions for accessing contiguous blocks of memory for every memory model in the obvious way.

memory_write_list :
  [Memory_struct → [Address, list[Byte] → [State → ExprResult[State, Unit]]]]

memory_read_list :
  [Memory_struct → [Address, nat → [State → ExprResult[State, list[Byte]]]]]

Our second base component, the semantics of data types, provides a suitable semantics for all C++ types and all necessary hardware data types (such as page-table entries). It exploits under-specification to make the detection of erroneous type casts and wrong implicit type conversions possible (like, for instance, reading data from a union with the wrong type) [2]. The data-type semantics is independent of any memory model. It provides operations to convert data from and to their object representation, which is of type list[Byte]. Writing and reading the object representation into and out of memory is done with the above functions memory_write_list and memory_read_list.

In our design, the three base components—hardware model, data types, and C++ semantics—are relatively independent of each other. It is therefore possible

- to add new operations to the hardware model,
- to use different versions of the hardware model for different parts of the hypervisor; the boot code of the hypervisor can, for instance, be verified against physical memory,
- to add additional axioms to the data types, e.g. to model compiler-specific assumptions about the size of some data types or the precise behavior of some type casts, and
- to adopt the semantics of new C++ features or compiler-specific C++ constructs.

Our hardware model and the C++ semantics are necessarily incomplete. Many of the omissions however do not lead to global assumptions on the validity of our verification. The hardware model, for instance, does not contain virtual 8086 mode, but the validity of our verification does not hinge on the absence of instructions that enable virtual 8086 mode. Instead the `VM` flag, which controls this mode, is protected

with a suitable side effect (see Section 4). Any attempt to enable virtual 8086 mode will yield a Fatal result. Hence a proof of normal termination suffices to show that virtual 8086 mode is never enabled. Similarly, the use of missing features in the C++ semantics will trigger an assertion in the semantics compiler.

For a number of features currently not present in our verification environment, we plan inclusion in the future. These features are (1) the Translation Lookaside Buffer (TLB)[6], (2) cache policy checking for devices, (3) segment offsets and segment size checking, (4) linking object code and instruction fetch to the abstract C++ semantics. Because of their absence we are currently unable to detect certain kinds of errors, namely

- TLB errors, e.g. inconsistencies between the TLB and the page tables, or implicit assumptions about the TLB size and structure,
- segment violations (the Robin micro-hypervisor uses a flat memory model where no segment violations can occur, however, currently we do not check that the segment descriptors are filled with the proper values),
- cache policy errors for memory-mapped devices, and delayed side effects for cachable memory-mapped devices,[7]
- discrepancies between our C++ semantics and the compiled object code, which (apart from compiler bugs) could occur for the following reasons: volatile-related errors in the source code[8], certain compiler optimizations (e.g. delayed write-back to memory), or self-modifying code (however, no self-modifying code is contained in our current verification target).

Once these missing features have been added, our verification will build on the following general assumptions:

- The software to be verified will be executed on a single-processor system.
- Caches for real memory are working completely transparent and can be ignored. This should be guaranteed by the hardware on single-processor systems.
- The involved software tools—the C++ compiler used to compile the Robin micro-hypervisor, our semantics compiler (including the Elsa C++ parser and type checker), and PVS—produce correct results.

3 The Plain-Memory Abstraction

The memory in an IA32 system is a sophisticated device: segments and page tables specify access rights, a given piece of memory might be visible in different virtual-address ranges, the address translation in the CPU from virtual to physical addresses might differ from what is specified in the page table because of bogus TLB entries,

[6] The TLB is a special CPU-internal cache for virtual-to-physical address translations.

[7] The source code that we currently target does not involve any devices. In general, cache policy checking for memory-mapped devices is trivial to add with our mechanism for side effects (see Section 4). To model cache effects on cachable devices, the model of the device should include the relevant cache effects.

[8] A C++ compiler is permitted to perform arbitrary optimizations with respect to non-volatile data. Memory accesses to such data are not part of the *observable behavior* of a C++ program, which makes a correct semantics difficult. At the moment our C++ semantics treats all data as volatile. A verification based on the current semantics will therefore not catch missing volatile annotations or missing memory fences.

and much more. We cannot ignore all these effects, not even for most innocent kernel code, because of the errors that they might cause.

As a consequence we designed the *plain-memory* abstraction for the verification of those parts of the kernel that require only the standard C++ memory model. It deals with the following issues.

- Writing or reading a single byte in memory can have devastating effects if one hits a memory-mapped device, a page table or simply an unmapped address. For correctness, the verification must therefore be carried out against a faithful model of IA32 memory. Plain memory provides a (comparatively) simple abstraction that can be used for those parts of the sources that only need well-behaved memory without special effects.
- The IA32 hardware provides several memory configurations: real-address mode, protected mode with and without paging. Our hardware model multiplies the number of different memories because we prefer to model different memory features (such as the TLB or execution of the page-fault handler) in different memory models. Most of the code does not depend on a concrete memory model and should consequently be verified against a suitable set of memory models. Plain memory permits precisely this because every memory model of interest will give rise to a model of plain memory.

Technically plain memory is a specification that provides byte-wise read and write access to memory, where special properties are guaranteed for *read-blessed* and *read/write-blessed* address regions. (There are additional operations to access side effects of memory-mapped devices and to enforce reserved bits. We ignore these things here, see Section 4 for details.) The general idea is simple. Memory at blessed addresses is sane: read access does not change anything in the blessed address range, and write access only changes the bytes written (in the expected way). Moreover, these special properties are maintained as long as only blessed addresses are accessed. No guarantees are made however for a memory access outside the blessed address regions. We have shown in PVS that these properties are satisfied by normal virtual memory (that is outside memory-mapped devices) under the following preconditions:

- All blessed addresses are mapped in the page table, and no two different read/write-blessed addresses are mapped to the same physical address. (Different read-blessed addresses might refer to the same physical address, and there might also exist other mappings for addresses outside the blessed address regions.)
- The page tables can only be read-blessed if the accessed and dirty bits are set. (This condition can be relaxed for page-table entries of read-blessed or unblessed memory.)

We want the plain-memory specification to be usable with all concrete memory models (including physical real-address memory). Therefore the specification must describe all properties only with the observations that can be made by reading and writing single bytes. In PVS the specification is split into a record of functions (capturing the plain-memory signature) and a predicate for the required properties. With this technique the axioms of the plain-memory specification do not show up as

axioms in the PVS formalization, hence they do not affect consistency. Instead, any use of a plain-memory property in a proof will spawn a subgoal requiring the proof of the plain-memory axioms. The plain-memory signature is defined as follows:

```
Plain_Memory : Type = [#
  mem : Memory_struct[State],        % see page 74
  states  :  PRED[State],            % states fulfilling the plain memory properties
  ro_addr  :  PRED[Address],         % read−blessed addresses
  rw_addr  :  PRED[Address]          % write−blessed addresses
#]
```

The properties of plain memory are specified as follows.

```
plain_memory?(pm) : bool =
   unchanged_memory_invariant?(pm‘mem, pm‘states,
      union( all permitted state transformers except write access to pm‘rw_addr ),
      union(pm‘ro_addr, pm‘rw_addr)) ∧
   unchanged_memory_invariant?(pm‘mem, pm‘states,
      memory_write_transformers(pm‘mem, pm‘rw_addr),
      pm‘ro_addr) ∧
   unchanged_memory_write_invariant?(pm‘mem, pm‘states, pm‘rw_addr) ∧
   changed_memory_invariant?(pm‘mem, pm‘states, pm‘rw_addr) ∧
   transformers_ok?(pm‘states,
      union( all permitted state transformers )) ∧
   side_effect_content_unchanged(union(pm‘ro_addr, pm‘rw_addr), pm‘states,
      memory_read_side_effect(pm‘mem)) ∧
   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
      memory_write_side_effect(pm‘mem))
```

We have omitted the involved expression for the union of all permitted state transformers. This set contains all read accesses to read- and read/write-blessed addresses, all write accesses to read/write-blessed addresses, and all corresponding side effects.

The first clause states that read accesses to blessed addresses and all possible side effects do not change the contents of any of the blessed addresses. The second clause expresses the same for write accesses and the read-blessed addresses. The third clause requires that a write access to one address leaves all other read/write-blessed addresses intact. The fourth clause states that write accesses actually change the written address in the right way. The utility predicates used in the first four clauses additionally require that the set of states forms an invariant with respect to the respective set of state transformers. This makes the plain-memory property an invariant: permitted state transformers must stay in the set of plain-memory states, in which all the nice properties hold. The fifth clause makes all memory accesses terminate with OK (which prohibits e.g. unhandled page-faults). The last two clauses require that side effects do not change the data read or written.

The plain-memory specification entails that only explicit writes change a memory cell. This property enables us to prove the following lemma.

```
plain_memory_read_write_other_res : Lemma
  plain_memory?(pm) ∧
  pm‘states(s) ∧
  in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧
  in_blessed_memory?(dt2, addr2, union(pm‘ro_addr, pm‘rw_addr)) ∧
```

```
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
valid_in_mem(pm,dt2)(addr2)(s)
  ⟹
    data( (write_data(pm,dt1)(addr1, data1) ##
           read_data(pm,dt2)(addr2))(s) )
      =
    data( read_data(pm,dt2)(addr2)(s) )
```

It expresses that for two variables of type dt1 and dt2 that lie disjoint in blessed memory, writing the first one does not change the contents of the second. This lemma is used in a rewrite engine that enables PVS to symbolically compute the value of a variable by going back to the last write access to that variable.

Our hardware model contains physical memory (RAM) as a base of all memory models. Physical memory provides one byte of storage for every address up to a certain maximum. Accesses above the maximum yield Fatal as result. Unsurprisingly we can prove that all states of physical memory form a plain memory, with all addresses below the maximum read/write-blessed.

Our model of linear memory contains page-table based address translation, but no TLB or page-fault handler yet. The linear memory is stacked on top of a plain memory using the general memory-structure interface (see Section 4.2 for more on the stacking of memory models). This plain memory is typically instantiated with physical memory (possibly containing devices). We have shown in PVS that the plain-memory properties are obtained for linear memory under the following preconditions:

- The code segment register (CS), determining the code privilege level, the page-table base register, and the page tables themselves remain unchanged (with the exception of access bits).
- Any translation for read or execute accesses succeeds for the entire blessed range of virtual addresses. Translations for writes succeed for the writable subset.
- Blessed writable virtual addresses map to blessed writable physical addresses, blessed read-only addresses map to blessed readable or writable physical addresses.
- Page tables reside in a memory area that is disjoint from the targets of the above mappings.
- There is no blessed shared-memory alias to a writable virtual address. (Virtual read-only regions may be shared arbitrarily.)

One point to highlight is that we only have to require those page-table entries to remain unchanged that are used in the translation of the virtual blessed address range. This allows us to modify unrelated page-table entries without loosing the blessing properties. We achieve this by requiring

```
disjoint?(virt_to_phys_range(s, union(pm‘ro_addr, pm‘rw_addr)),
          address_in_pt_range?(s, union(pm‘ro_addr, pm‘rw_addr))),
```

where virt_to_phys_range translates all addresses in the virtual blessed address range, and address_in_pt_range? returns the corresponding physical addresses containing the page-table entries for this range.

4 Memory-Mapped Devices and Reserved Bits

Although most device drivers reside outside the micro-hypervisor, some devices (e.g. the interrupt controller) must remain under kernel control to prevent malicious code from monopolizing the system. To program these devices, the micro-hypervisor code accesses certain device registers. Unlike normal RAM, these registers show very special behavior when accessed. Special-purpose processor registers (such as the IA32 control registers [IA32-3a-2.5] [9]) are similar to device registers in that read or write accesses to them may cause special effects. For our verification attempt, the following special effects are important.

Reserved bits The value of reserved bits must not be modified, or otherwise the processor behavior is undefined. For example, bits 0–2 and bits 5–11 of the IA32 page-table base register (`CR3`) are reserved [IA32-3a-2.5].

Access type restrictions Some device registers are read-only respectively write-only accessible, or they allow no instructions to be fetched. ROM is a prominent example of a read-only accessible device.

Alignment restrictions Some devices require that registers are accessed only at certain offsets relative to the register base address. Furthermore, each access must read or write a certain amount of data at once. For example, the registers of the IA32 advanced programmable interrupt controller (APIC) are aligned on 16-byte boundaries. They must be accessed with 4-byte wide and 4-byte aligned reads and writes [IA32-3a-8.4.1].

Side effects Reading or writing causes side effects on some devices. For example, writing to the IA32 APIC *end of interrupt* register signals completion of the interrupt-handling procedure [IA32-3a-8.8.5]. This may cause the immediate delivery of the next pending interrupt.

More abstractly, we can summarize these behaviors as follows: reading and writing certain registers and certain locations in memory may result in modifications to the system state, to the memory (or register) contents, and to the value read or written. Furthermore, the behavior of an operation may be undefined; in this case the verification should fail.

4.1 Modeling Devices and Reserved Bits

Instead of modeling memory and additional devices as parallel abstract machines, we prefer the following approach. We extend the Memory_struct of each memory model with two *side-effect transformers* memory_read_side_effect and memory_write_side_effect of the following type:

$$\mathsf{Address},\ \mathsf{list[Byte]},\ \mathsf{bool} \to [\mathsf{State} \to \mathsf{ExprResult[State,\ list[Byte]]}].$$

Unlike memory_read and memory_write, these side-effect transformers take the entire list of bytes read from (or written to) the given address. This is necessary to enforce alignment restrictions, which require knowledge about the amount of data that is

[9] The notation [IA32-3a-2.5] refers to Volume 3a, Section 2.5 of the *Intel 64 and IA-32 Architectures Software Developer's Manual* [4].

read (respectively written) at once. The third parameter is an indicator whether the access crossed a page boundary in a higher virtual memory layer. We elaborate on the use of this parameter in Section 4.2.

In the following we use a simple memory-mapped random number generator (RNG) to illustrate how one can model side effects and access restrictions. The RNG device provides one read-only memory-mapped register, rnd_val, that contains an unspecified (supposedly "random") value. The internal state of the device (which is kept in addition to the memory state) contains two natural numbers seed and access_count. The value of access_count is incremented as a side effect with each memory access. The seed is left unspecified.

To obtain the random value we apply a completely unspecified function random to the seed and the current access_count. Because access_count is strictly increasing, we get potentially different values for every access to rnd_val. Under-specified and non-deterministic behavior of more complicated devices can be modeled in a similar fashion.

4.1.1 Access Type and Alignment Restrictions

Modeling devices with access-type and alignment restrictions is straightforward by checking these restrictions for overlapping accesses. As an example, we impose that rnd_val must be accessed with machine-word granularity (as `unsigned int`).

```
unaligned_access(a : Address, bl : list[Byte], cp : bool) :
    [Random_device_memory → ExprResult[Random_device_memory, list[Byte]]] =
  If disjoint?(address_block(a, length(bl)), address_block(rnd_val, size(uidt(dt_uint)))) ∨
    (¬ cp ∧ length(bl) = size(uidt(dt_uint)) ∧ a = rnd_val)
  Then ok_result(bl)
  Else fatal_result Endif
```

Here size(uidt(dt_uint)) comes from the C++ data-type model and gives the number of bytes of an `unsigned int` (which is usually 4 on the IA32 architecture, although this may vary with the C++ implementation). To require the read-only behavior of rnd_val we use the following after the above alignment check has been passed:

```
write_rnd_dev(a : Address, bl : list[Byte], cp : bool) :
    [Random_device_memory → ExprResult[Random_device_memory, list[Byte]]] =
  If a = rnd_val
  Then fatal_result
  Else ok_result(bl) Endif
```

The memory_read_side_effect and memory_write_side_effect transformers of the RNG device are set to compositions of the above two transformers and other checks, which we discuss below.

4.1.2 Reserved Bits

Reserved-bit restrictions come in two flavors, depending on whether the value of reserved bits is specified. For example, the IA32 processor manuals [IA32-3a-2.5] specify bits 11–31 of the `CR4` register as reserved. The value of these bits must be 0. As long as the value of reserved bits is specified, we merely have to check that the value in the byte list passed to the memory_write_side_effect transformer is according to the specification. Registers whose reserved bits all have specified values can then be modified simply by writing to these registers.

Other special-purpose processor registers and device registers leave the value of certain reserved bits undefined. In this case we match against an unspecified value in the memory_write_side_effect transformer. Because the initial register contents are not specified, we can establish that reserved bits are unaltered by a write access only when the written data originates from a previous read of this register.

Reserved bits can also be used to restrict the processor modes in which the micro-hypervisor may execute. For example, we fix the mode bits in the IA32 control registers CR0 and CR4 to the setting for 32-bit paged, protected mode. This prevents the kernel from switching back to real mode. Consequently it suffices to model only those parts of real mode that are required for the verification of the kernel's boot-strapping code.

4.1.3 Side Effects

Side effects on reads (respectively on writes) cause additional parts of the system state to be updated. Here, one can either update the memory state itself, or add an additional device state. For the random number generator, we decided to use the latter approach. As described earlier, our RNG device implements a side effect to count all memory accesses in its internal state.

```
access_count(a : Address, bl : list[Byte], cp : bool)(s) :
                                ExprResult[Random_device_memory, list[Byte]] =
  OK(increase_access_count(s)(length(bl)), bl)
```

Another side-effect transformer generates the "random" (i.e. unspecified) value when the rnd_val register is read.

```
random : [nat, nat → { l : list[Byte] | length(l) = size(uidt(dt_uint)) }]

read_rnd_val(a : Address, bl : list[Byte], cp : bool) :
     [Random_device_memory → ExprResult[Random_device_memory, list[Byte]]] =
  If a = rnd_val
  Then λ(s : Random_device_memory) :
                                ok_result(random(seed(s), access_count(s))(s)
  Else ok_result(bl) Endif
```

4.2 Stacking Memory Layers

For greater modularity we split different hardware features into separate memory models where possible. For instance, segment based and page-table based address translation (which are both part of the virtual memory in the IA32 architecture) can nevertheless be split into two independent memory models. To obtain the overall effect we stack different memory models, exploiting the general memory_struct interface. Every memory layer performs its functionality before invoking the underlying layer via the abstract interface. The bottom layer is a model of physical memory that contains a byte array to store the memory contents.

A good example of the stacking of memory layers is provided by linear memory, which adds page-table based address translation to an underlying physical memory. In order to keep the stacking flexible, the linear memory is not based on our model of physical memory directly, but is instead parameterized with an arbitrary plain-memory model pm. (This model can be instantiated with physical memory, and with

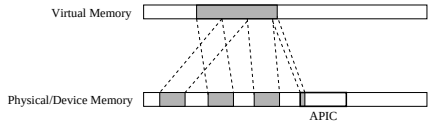

Fig. 3. Splitting of side effects when stacking virtual memory on top of device memory.

any other memory model that satisfies the plain-memory properties.) The linear memory layer defines page-table based lookup and address translation as described for 32-bit page tables in [IA32-3a-3.7]. Finally it defines functions for filling the memory_struct of the linear memory model. The function for memory_read is given by

```
linear_read(a : Address) : [Linear_memory → ExprResult[Linear_memory, Byte]] =
  If in_memory(max_linear)(a) Then
    If Mem?(type_of(a)) Then
      linear_resolve(a, Read) ## λ(pa : Address) : memory_read(pm‘mem)(pa)
    Else
      memory_read(pm‘mem)(a)
    Endif
  Else fatal_result Endif
```

The state-space type Linear_memory is equal to the state space of the plain-memory parameter pm. The function linear_read first checks if the access is within the memory or register bounds. For a real memory access, the virtual address a is then translated into a physical address pa, which is used to access the underlying memory of pm. Any abnormal result of the address translation (because of page faults or data-type errors in a page table) is propagated to the outside via the composition of state transformers, ##. Register accesses are passed to pm without address translation, of course.

Our random number generator from the previous section was also implemented as an independent memory layer, which adds the functionality of the RNG device to the underlying memory layer.

However, when combining virtual memory with memory-mapped devices, one must deal with new problems. The virtual memory performs address translation and may thereby split an access to a contiguous block of memory into several accesses to noncontiguous blocks. By coincidence the splitting might result in an access to a memory-mapped device that seemingly satisfies all alignment and granularity requirements of the device. For instance, in Figure 3 the very last piece of the virtual address block is mapped precisely to the first APIC register. We view such an access as an error, because it is only part of a larger memory access. Moreover, the IA32 architecture gives no guarantees that in the case depicted in Figure 3 the access to the APIC is performed with 4-byte granularity [IA32-3a-7.1.1], as required by the APIC.

To detect this kind of error, we introduce a crossed-page indicator cp as argument to our side-effect transformers. Initially being false, this indicator is set to true when the address translation splits a contiguous memory access. The side effect transformer unaligned_access (see page 81) of our RNG device always checks the cp flag and delivers an error if it is true.

5 Example Verification

To illustrate how kernel-code verification works in our environment, we have proved two partial correctness properties of a simple linear search (in an array of N unsigned integers) in PVS. The C++ implementation of the search algorithm uses pointers and pointer arithmetic:

```
unsigned int a[N], value;
unsigned int *first = &a[0];
unsigned int *last = &a[N];
unsigned int *current;

current = first;
while(current < last) {
  if (*current == value) break;
  current++;
}
```

More precisely, we have shown that the current pointer refers to an array element containing the search value if the value is contained in the array, and to the element one beyond the array bounds (last) if no such element is present. The verification of these properties proceeds according to the approach that was outlined in Figure 2 on page 73. First, (1) the C++ sources for the search program are translated into their semantics in PVS. Second, (2) the correctness properties are formulated as pre- and postconditions, and then verified against the plain-memory specification. Our verification thus shows that, under suitable assumptions, the example program runs correctly both in physical and virtual memory. Finally, to avoid vacuous results, (3) the plain-memory preconditions were validated for concrete stacks of memory models (e.g. linear memory on top of the RNG device on top of physical memory).

Because we have established parts (2) and (3) separately, changing the underlying hardware/memory model only requires that one repeats the validation of the plain-memory assumptions (part (3)) for the new memory model.

5.1 C++ to Semantics Translation

The semantics compiler translates the above C++ code into its PVS semantics. Expression-to-statement and lvalue-to-rvalue conversions are made explicit. We only show the translation of the second part (lines 6–10) of the above code snippet.

```
e2s[State, Address, Semantics_void]
    (assign(pm, dt_pointer)(id(current), l2r(pm, dt_pointer)(id(first)))) ##
while(l2r(pm, dt_pointer)(id(current)) < l2r(pm, dt_pointer)(id(last)),
  if_else(
    l2r(pm, dt_uint)(deref(pm)(l2r(pm, dt_pointer)(id(current)))) == literal(value),
    break,
    skip) ##
  e2s[State, Semantics_pointer, Semantics_void](postinc(pm)(id(current)))),
```

Here first, last and current are addresses of disjointly allocated pointer variables. Currently we require disjointness of these variables in a precondition, but once the formalization of memory allocation is complete, disjointness will be derived from the allocator model.

5.2 Verification Against the Plain-Memory Abstraction

Verification is currently done by automatic loop unrolling and simplification according to the plain-memory rewriting rules. For this, the necessary preconditions in_blessed_memory and valid_in_mem for not previously written variables have been added to the precondition. in_blessed_memory states that the variable is correctly allocated in blessed memory, valid_in_mem that the memory contains a valid bit representation for the variable. Typically the latter is established by a previous write to this variable.

In our C++ semantics, all expressions and most statements are expressed using a combination of only four different state transformers: read_data, write_data, ok_result(data) (which returns OK(s, data)), and fatal_result (which produces Fatal). It is therefore possible to simplify expressions by first expanding them to sequences of these transformers, and then simplifying these sequences using the plain-memory rewriting rules (e.g. the rule in Section 3).

Similarly, we evaluate statements up to the point where only expressions remain in the code sequence. For example, under the precondition OK?((expr ## b_ex)(s)) the sequence

(e2s(expr) ## if_else(b_ex, stmt_if, stmt_else))(s)

is rewritten into

(e2s(expr ## b_ex) ## **If** data(expr ## b_ex)(s) **Then** stmt_if **Else** stmt_else **Endif**)(s)

This simplifies with an appropriate rewriting rule for e2s to either expr ## b_ex ## stmt_if or to expr ## b_ex ## stmt_else, depending on the value of data((expr ## b_ex)(s)). Likewise, we rewrite the statements stmt_if and stmt_else to expression sequences containing only read_data, write_data, and the above result transformers. Because of this transformation it suffices to define and prove the plain-memory rewriting rules only for data reads and writes. All other rules of the rewriting system operate independently from the data-type or memory model.

5.3 Establishing Plain Memory

For each memory model we established the plain-memory property for a certain range of addresses. As stated above, for physical memory this is the entire address range. Stacked models contain preconditions which require the blessed address range to be contained in the blessed range of the underlying memory model. It is therefore sufficient to show that the addresses used in the code to be verified all reside in blessed memory. Accesses outside the blessed-memory address range automatically violate the plain-memory assumption and cannot be simplified with the plain-memory rewriting rules. In such a case the plain-memory property must be reestablished before one can proceed with the automatic simplification.

For our verification example, one needs the following preconditions:

- The variables are allocated so that they do not overlap with the registers of the random device.
- The variables are allocated so that they do not overlap with a page-table entry

used to access some of the variables.

- All page-table entries of these variables are writable (because reference bits may be written back to memory).
- The memory of current is not virtually aliased with any of the other variables used in the search program.

Note that it is possible to have virtual aliases in the array or for the first and last pointers, as these are read only.

6 Conclusions

In this paper we have presented two details of our approach to model the memory of an IA32 system. The first detail is a specification of well-behaved *plain memory*. This specification allows us to maintain an abstract level of reasoning with reasonable efficiency on top of a complex model of paged virtual memory. The second detail is our modeling of memory-mapped devices and reserved bit restrictions. We use side-effect state transformers that are performed before and after memory access to uniformly model both, reserved bits and memory-mapped devices. As demonstrations we have included the formalization of a (memory-mapped) random number generator, and an example verification of a simple C++ code fragment.

Related Work. There have been extensive attempts to reconcile the untyped memory model of C with a typed view, see e.g. [10]. At the other end of the spectrum, complete micro-processors have been formally verified at the gate level [1]. Our current work is located in-between those efforts. We take a correct IA32 processor for granted. Our aim then is to establish that the view on memory provided by the Robin micro-hypervisor, despite various peculiarities present in the architecture's hardware (virtual vs. physical memory, memory-mapped devices, etc.), is a model of C++ memory, which is well-behaved in the sense that one does not need to worry about low-level features like virtual address aliasing anymore. Perhaps most closely related is the verification of page table algorithms in [9], which still uses a rather abstract memory model however.

Acknowledgments We would like to thank the anonymous referees for their valuable suggestions.

References

[1] Beyer, S., C. Jacobi, D. Kröning, D. Leinenbach and W. J. Paul, *Putting it all together—formal verification of the VAMP*, STTT Journal, Special Issue on Recent Advances in Hardware Verification **8** (2006), pp. 411–430.

[2] Hohmuth, M. and H. Tews, *The semantics of C++ data types: Towards verifying low-level system components*, in: D. Basin and B. Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003). Emerging Trends Proceedings* (2003), pp. 127–144, technical Report No. 187.

[3] Huisman, M. and B. Jacobs, *Java program verification via a Hoare logic with abrupt termination*, in: T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science **1783** (2000), pp. 284–303.

[4] Intel Corporation, Denver, CO, "Intel 64 and IA-32 Architectures Software Developer's Manual," (2007), order Number: 25366[5-9]-023US.

[5] Owre, S., S. Rajan, J. Rushby, N. Shankar and M. Srivas, *PVS: Combining specification, proof checking, and model checking*, in: R. Alur and T. Henzinger, editors, *Computer Aided Verification*, Lecture Notes in Computer Science **1102** (1996), pp. 411–414.

[6] Tews, H., *Micro hypervisor verification: Possible approaches and relevant properties*, in: *NLUUG Voorjaarsconferentie 2007: Virtualisatie*, 2007, pp. 96–109.

[7] Tews, H., *Olmar: manipulating C and C++ abstract syntax trees in OCaml*, in: H. Tews, editor, *Proceedings of the C/C++ Verification Workshop*, 2007, pp. 103–113, technical report ICIS-R07015, Radboud University Nijmegen.

[8] Tews, H., B. Jacobs, E. Poll, M. van Eekelen and P. van Rossum, *Specification and verification of the Nova microhypervisor* (2007), deliverable D.6 of the Robin project, available at http://www.cs.ru.nl/~tews.

[9] Tuch, H. and G. Klein, *Verifying the L4 virtual memory subsystem*, in: G. Klein, editor, *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, NICTA Technical Report 0401005T-1 (2004), pp. 73–97.

[10] Tuch, H., G. Klein and M. Norrish, *Types, bytes, and separation logic*, in: M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, 2007, pp. 97–108.

Specification of Conditions for Error Diagnostics

Christof Efkemann[1,2]

TZI
University of Bremen
Bremen, Germany

Tobias Hartmann[1,3]

TZI
University of Bremen
Bremen, Germany

Abstract

This paper describes the basic concepts of error diagnostics and an associated rule system whose application helps to identify potential hardware/software locations of errors which caused a failure observed while executing tests of embedded systems. Further on, an overview over the "Avionics Smoke Detection System" is given, where the main algorithms have been applied. The methods, techniques and tools described here rely on the preceding investigations performed with respect to signal flow and the observation of causal chains in distributed test benches during test suite executions. These results have been elaborated in the KATO project within the German aerospace research programme LUFO III.

Keywords: Embedded systems, error diagnostics, RT-Tester.

1 Introduction

This document describes the results achieved by the University of Bremen, TZI, for KATO-TP13 work packages "*Specification of conditions for error diagnostics*" and "*Implementation of an error diagnostic prototype*". With almost ten years of experience in the area of test and verification of avionics controllers all kinds of faults, errors and anomalies have been observed and detected by our workgroup. The obvious next step is to classify the encountered faults. Based on the fault models commonly used in semiconductor fault diagnostics[1][6] we applied these models and methods and provide a complete strategy for locating faults in distributed embedded

[1] The authors have been partially supported by the BIG Bremer Investitions-Gesellschaft under research grant 2INNO1015B

[2] Email: chref@tzi.de

[3] Email: hartmann@tzi.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

systems within their environments. The methods, techniques and tools illustrated here rely on preceding investigations, which have been documented in [7].

Due to legal constraints we cannot present "real" source code, but instead we illustrate our methods for error diagnostics with a slightly simplified and abstracted example of a real system. The System Under Test (SUT) and its testing environment are depicted in Fig. 1 and introduced in the paragraphs below.

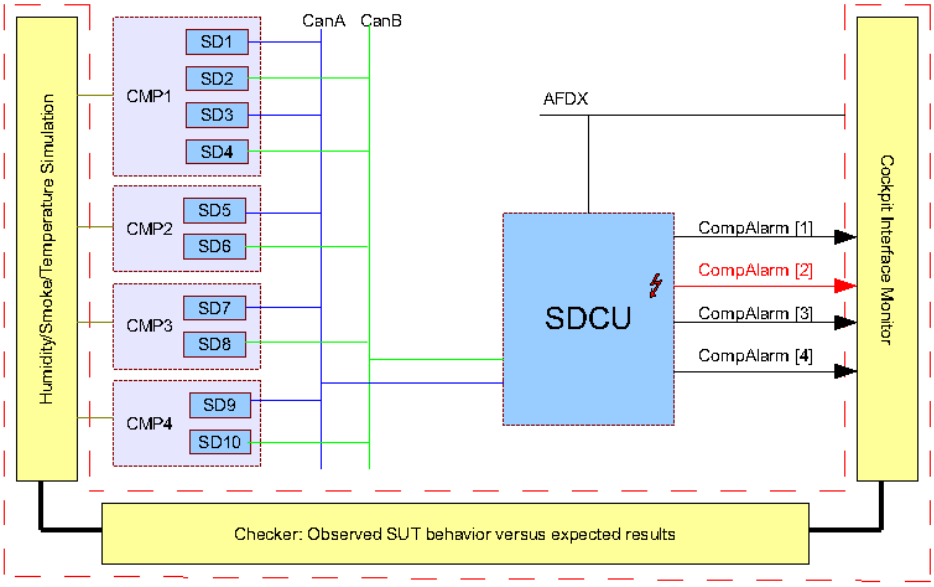

Fig. 1. Testing scenario

A simplified version of an aircraft *smoke detection system* is used as the SUT. Its model, which is assumed to be correct, and the corresponding faulty implementation are described in Section 2. The central control unit (*smoke/fire detection control unit (SDCU)*) contains an error similar to one of those errors found during "real" controller testing.

The SDCU is connected to the cockpit via a bus system called Avionics Full Duplex Switched Ethernet (*AFDX*), which is commonly used in modern aircraft. This bus is used to send the alarm states of the compartments to the cockpit. These messages are pictured as arrows, leading from the SDCU to the cockpit.

However, to emphasize the fault, the reporting message is made visible in the test scenario overview. The faulty part is marked by a red arrow.

To keep the overview short and simple, the flow of the messages from the SDCU to the sensors – and vice versa – are not shown. They are routed within the two CAN buses A and B.

The implementation is embedded into a testing and simulation environment, using Verified's RT-Tester tool[12]. The resulting testing environment and the test suite executed therein are described in Section 3. In Section 4, the application of the diagnostic tool kit described in the preceding chapters is illustrated for the purpose of identifying the causes and locations of the test failures observed.

2 SDF SUT Model and Implementation

The system consists of (see Fig. 1 above) the following items:

- The **smoke detectors (SD)** measure humidity, smoke and temperature at designated locations in the aircraft and transmit smoke/fire alarms and other status information on a CAN bus system.
- The **smoke detection control unit (SDCU)** collects sensor data from various busses and distributes alarms and other messages to the cockpit (and other systems).
- The **communication media** consist of
  · CAN busses for SD ↔ SDCU communication,
  · AFDX bus for SDCU ↔ Cockpit communication,
  · discrete I/O lines for cockpit indications.

Observe that real smoke detection systems use several additional communication interfaces, in particular, aural and visual indication to the aircraft cabin. For the purpose of illustrating our concepts for error diagnostics, however, these interfaces and their associated additional functionality are not relevant. The central SUT component is the SDCU controller. One of the most important SDCU methods is `processMessages()`. Here, the decisions whether to raise compartment smoke alarms are made. In the correctly modelled system the alarm status of compartment c depends on the conjunction of local alarm states. For compartment Comp1 the states of the sensors SD1 to SD4 are evaluated. Two sensors are used for the compartments Comp2 to Comp4 respectively. This leads to the following alarm equations:

- Comp1:
  $$compAlarm[1] = (sensorSDSAlarm[1] \wedge sensorSDSAlarm[2]) \vee (sensorSDSAlarm[3] \wedge sensorSDSAlarm[4])$$
- Comp2 ... Comp4:
  $compAlarm[c] = sensorSDSAlarm[2c+1] \wedge sensorSDSAlarm[2c+2]$,
  $c \in \{2 \dots 4\}$

However, the SUT implementation contains an error: For compartment Comp2, the alarm decision depends on $sensorSDSAlarm[5] \wedge sensorSDSAlarm[13]$ instead of $sensorSDSAlarm[5] \wedge sensorSDSAlarm[6]$. In the diagnostic procedure described below, this failure will be referred to as *Failure*.

3 SDF Testing Environment

In this document we focus on *system integration testing*, so the tests will typically be

- **Black-box tests** on controller level: Since controllers have been tested by their suppliers during the mandatory HW/SW integration test suite, the system tests will deal with controllers as black-boxes.
- **Black-box tests** on peripheral device level (same reason as for controllers).

- **Grey-box tests** with respect to inter-controller communications: During system integration testing, network monitors are typically available to record at least a portion of the data exchange between communicating controllers. For example, AFDX monitors and ARINC 429 monitors can be used to record snapshots or specific types of data packages. Some communications, however, often cannot be observed, such as, for example, the data exchange between redundant fault-tolerant controllers.
- **Grey-box tests** with respect to controller ↔ device communication: Just like inter-controller communications, the data exchange between controllers and their directly connected peripherals can be – at least partially – observed using bus monitors (e. g. monitors for CAN busses, ARINC 429 busses), or measurement equipment (e. g. for discrete and analogue I/O interfaces).

The RT-Tester tool can be used for all kinds of tests – from unit level tests to hardware/software integration tests (described in [5]). However, in this document we focus on system integration tests, and all mentioned tests were executed using this tool.

4 Diagnostic Procedure

4.1 Step 1: Initial Black-Box Test Results

A diagnostic procedure is always triggered by a discrepancy observed during a test suite. The failure – contained in the SDCU itself – is detected during a functional test of compartment smoke alarms in compartment Comp2.

The initial values for the compartments are shown in the first test log. The values for humidity and smoke are recorded as percent values. The temperature is given in degrees Celsius.

```
TM 00027023005 AM   4 0 : ( 25 ) ENVIRONMENT:
                        : ---------------------------
                        :     COMP:   1    2    3    4
                        : HUMIDITY:   5    5    5    5
                        :    SMOKE:   2    2    2    2
                        :     TEMP:  20   20   20   20
```

The test environment simulates a relevant smoke and slight temperature increase for Comp2, which is recorded in the test execution protocol as

```
TM 00029035001 AM   4 0 : ( 27 ) ENVIRONMENT:
                        : ---------------------------
                        :     COMP:   1    2    3    4
                        : HUMIDITY:   5    5    5    5
                        :    SMOKE:  90   90   90    2
                        :     TEMP:  30   30   30   20
```

The threshold values for alarms are at 80% for smoke and 65° for temperature. Therefore an alarm must be reported for three different compartments: Comp1, Comp2 and Comp3. However, no alarm is reported for the compartment Comp2:

```
TM 00034035267 AM   4 0 : ( 32 ) SDCU:
                        : --------------------
                        :      COMP: 1  2  3  4
                        :     ALARM: 1  0  1  0
```

This test step shows that after 5 seconds (timestamp unit: microseconds), there is still no smoke alarm for compartment Comp2, leading to test failure.

4.2 Step 2: Interface Analysis – Generation of the Causal Graph – Version 1

The causal graph shows all components and interfaces that may possibly influence the values passed along interface `compAlarm[2]`. For the version 1 representation of the graph all components which are *physically* connected are considered, because bridging faults[4] in components may result in arbitrary data flows, as long as a physical connection is available. As soon as certain types of internal component faults can be excluded, the causal graph may be narrowed, leading to new versions. Conversely, a more detailed analysis of a potentially faulty component may require to analyse its *internal data flow*. This leads to a *refinement* of the causal graph version n, where a component node C in version n is replaced by a detailed data flow network representing communications within C and leading to a new version $n+1$ of the graph. The initial version of the causal graph for the failure detected on interface `compAlarm[2]` is shown in Fig. 2. The graph contains cycles, because messages are transported from smoke/fire detectors to the SDCU and vice versa, and the messages from SDCU to SDs can also influence the state on interface `compAlarm[2]`: For example, if the SDCU would fail to poll sensors SD5 and SD6, then no alarm messages would be sent from SD5 and SD6 to the SDCU.

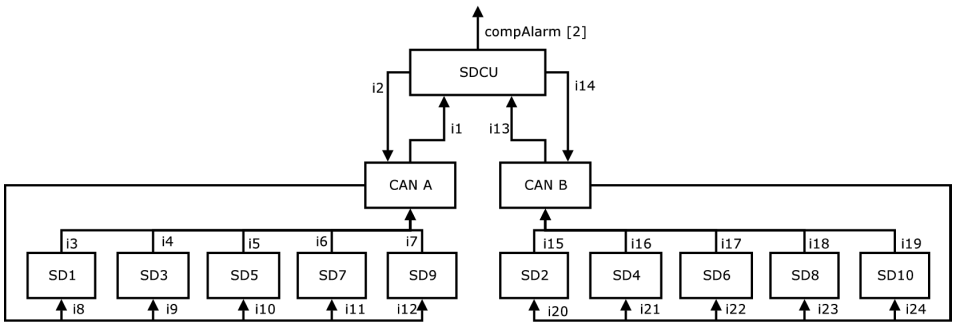

Fig. 2. Causal graph, version 1, for Failure

4.3 Step 3: Interface Analysis – Generation of the Fault Tree

The fault tree[8][11, pp. 43ff] constructed in Step 3 depicts the possible error hypotheses, together with the boundary conditions which must hold in order to make a hypothetical error cause the observed failure on the interface between SUT and testing environment. The error classification used for each component follows the fault models introduced in [7, pp. 13ff] and the fault tree construction technique described in [7, pp. 6ff]. In the diagnostic procedure described here we omit the

[4] For integrated circuits this is discussed in [10, pp. 335ff]

possibility of an external intruder (see [7]) because we are dealing with a closed system whose components are well-known.

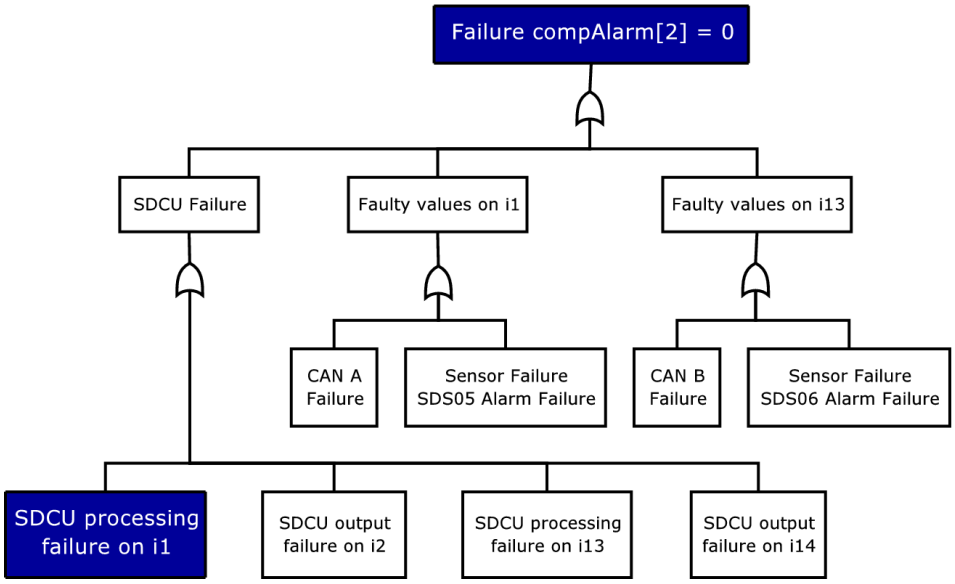

Fig. 3. Fault tree associated with interface analysis for Failure

Following the paths in the causal graph of Fig 2, we can immediately derive the first levels of the fault tree, as depicted in Fig. 3: The failure on output interface `compAlarm[2]` might be directly caused by an internal SDCU failure or by an erroneous input on the interfaces *i1*, *i13* between SDCU and the CAN bus. SDCU-internal failures may either cause faulty transformations from *i1*- and *i13*-inputs to `compAlarm[2]` outputs or by output faults on interfaces *i2* and *i14*, which may in turn cause unexpected smoke detector behaviour, so that the alarm messages of the crucial detectors SD5, SD6 are either not sent at all or not accepted by the SDCU. Faulty input values to a correctly operating SDCU may either be caused by a failure of the CAN bus or by sensor failures. Since compartment alarms indicated on output `compAlarm [c]`, $c \in \{1 \dots 4\}$ require that both [5] associated sensors SD(2c+1), SD(2c+2) signal an alarm, the crucial sensors for the observed failure on `compAlarm[2]` are SD5 and SD6.

Refining the possibilities of SDCU processing failures on interface *i1* and *i13* leads to fault sub-tree 4: Not showing fault-types which are *a priori* highly improbable leaves us with 5 potential types of fault (as classified in [3, chap. 7]):

(i) **Bridging faults** would arise if – due to internal interface errors of the SDCU – the `compAlarm[2]` output could be influenced by the values of other inputs which should be disregarded in a correctly operating SUT: If the SDCU used the wrong sensors (SD1,2,3,4,7,8,9,10) to determine the output on `compAlarm[2]` and these sensors were not in alarm state, then no compartment alarm would be raised on `compAlarm[2]`.

(ii) **Signal deletion errors** would arise if the *i1* or *i13* input would not be relayed internally to the SDCU sub-component responsible for raising the

[5] Comp1 is an exception: there are four sensors installed

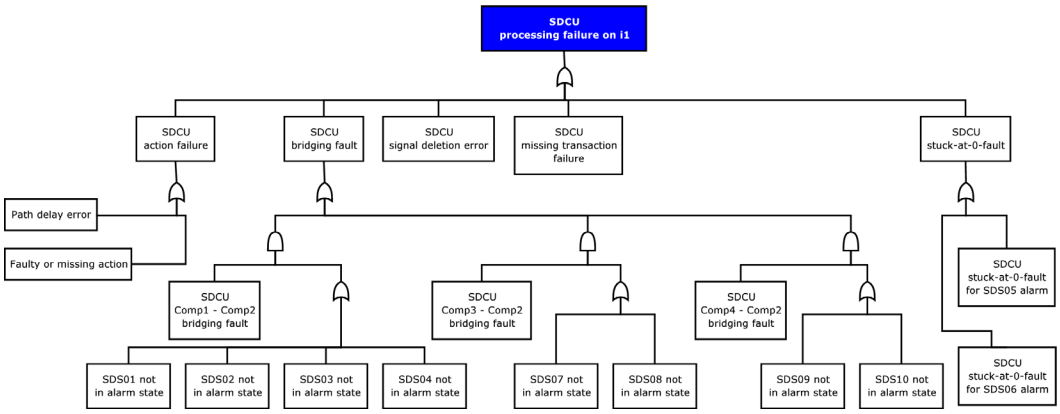

Fig. 4. Fault sub-tree associated with potential SDCU processing failures on interface *i1* which might cause Failure

`compAlarm[2]` compartment alarm.

(iii) **Stuck-at-0 faults** occur if the guard condition for raising `compAlarm[2]` to 1 were faulty, so that it always evaluates to *false*.

(iv) An **action failure** would arise if the assignment of the new value 1 to `compAlarm[2]` were faulty.

(v) A **missing transition** failure would arise if inputs from sensors SD5 and SD6 on interface *i1*, *i13* would simply not trigger any action at all.

Observe that the fault tree can be set up incrementally, so that extensions of leaves are only performed after other branches of the tree have been pursued which did not help to uncover the failure under investigation. We will therefore analyse the potential SDCU failures further, before refining the other branches of the initial fault tree 3.

4.4 Step 4: Falsification of Fault Hypotheses

After the fault tree associated with the interface analysis has been elaborated, the fault hypotheses implied by nodes of the tree can be falsified one by one, until the cause for the observed failure has been identified. Falsification can be performed by

(i) analysing additional data recorded during the test,

(ii) performing additional system integration tests designed to verify/falsify a specific fault hypothesis,

(iii) performing additional lower-level tests (e. g. HW/SW integration tests or unit tests), in order to observe additional interfaces which were not visible on system integration test level,

(iv) code execution or interpretation using concrete values,

(v) abstract code interpretation using interval values (like described in [2],

(vi) informal or formal code analysis (inspection, formal verification etc.).

Obviously, method 1 is the most preferable one since it neither requires additional tests nor source code availability. Following the fault model from [7], it will now be analysed how specific tests can be used to falsify fault hypotheses associated

with a given type of error.

4.4.1 Falsification tests for bridging faults.
Bridging faults imply that unintended inputs $x_i, i = 0, 1, 2, \ldots$ to the erroneous component influence a certain output y in an illegal way. A falsification test therefore consists of systematically changing the inputs x_i and observing whether a certain combination can illegally stimulate output changes on y. For the potential bridging faults of the SDCU as depicted in Fig. 4 the interpretation is as follows, as can be seen from the fault tree:

- If the observed error is due to a bridging fault from Comp1 sensor values to Comp2 sensor values, then it can only arise if sensors SD1,2,3,4 are *not* in ALARM state: Otherwise we would observe a correct output `compAlarm[2]=1`, though illegally caused by the wrong sensors SD1,2,3,4.
- Similarly, if the observed error is due to a bridging fault from Comp3 sensor values to Comp2 sensor values, then it can only arise if sensors SD7,8 are *not* in ALARM state.
- Analogously, if the observed error is due to a bridging fault from Comp4 sensor values to Comp2 sensor values, then it can only arise if sensors SD9,10 are *not* in ALARM state.

Analysing the test execution log, we see that when the output failure `compAlarm[2]=0` is observed, sensors SD1,2,3,4 are already in state ALARM, and the Comp1 alarm has been indicated correctly. As a consequence, there can be no bridging fault from Comp1 sensor values to Comp2 sensor values. Also, the sensors SD6,7 are in state ALARM and the compartment alarm for Comp3 is displayed: there can be no bridging fault from Comp3 to Comp2 either. The simulated environment for all compartments show that the thresholds for smoke are exceeded in the compartments Comp1, Comp2 and Comp3 (see log at timestamp 00029035001 μs, p. 91). The log at timestamp 00034035267 μs (see p. 92) shows that no alarm was detected for Comp2.

```
TM 00029024572 AM   4 0 : ( 27 ) SENSORS SDS:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  1  1  1  1  0  0
                        :    DEFECT: 0  0  0  0  0  0  0  0  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  0  0  0  1  1
```

Unfortunately, the sensors SD9,10 were still in state STANDBY when the failure occurred, so the test run could not exclude the bridging fault from Comp4 sensor values to Comp2 sensor values: We need an additional test where the sensors SD9,10 are permanently in state ALARM. If this does not lead to an output `compAlarm[2]=1`, this also excludes the second potential bridging fault.

Test: Bridging

In this test, the environment for compartment Comp4 is set in a way that the installed sensors will report an alarm. This can be read from the execution log at timestamp 00029028720 μs.

```
TM 00029028720 AM   4 0 : ( 27 ) ENVIRONMENT:
                        : ---------------------------
                        :     COMP:   1    2    3    4
                        : HUMIDITY:   5    5    5    5
                        :    SMOKE:  90   90   90   90
                        :     TEMP:  30   30   30   30
```

The Sensors SD9,10 are set to state alarm, which is reported at timstamp 00029029193 μs.

```
TM 00029029193 AM   4 0 : ( 27 ) SENSORS SDS:
                        : ------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  1  1  1  1  1  1
                        :    DEFECT: 0  0  0  0  0  0  0  0  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  0  0  0  0  0
```

As expected, the SDCU sets the alarm for compartment Comp4.

```
TM 00029029894 AM   4 0 : ( 27 ) SDCU:
                        : -------------------
                        :     COMP: 1  2  3  4
                        :    ALARM: 1  0  1  1
```

As shown in this test, the occurrence of a bridging fault can be excluded.

4.4.2 Falsification tests for signal deletion errors.

Systematic tests for falsification of signal deletion errors can be designed and executed if the SUT has the following proven property:

> If signal value v_0 is correctly transmitted over interface i, then value v_1 must be correctly transmitted, too.

Such a hypothesis is fulfilled[6], for example, if i is a shared variable interface and all messages placed into i have the same length and a checksum, allowing to detect corrupted (in particular, truncated) messages.

If the hypothesis holds, an additional test stimulating v_0 can be executed. If it passed then this implies that v_1 was not lost within the SUT. In our case, the hypothesis is fulfilled by the SUT implementation. Therefore an additional test can be created where another state value of sensors SD5,6 is stimulated, for instance with a FAILURE state. When the sensors transmit the state *failed*, the SDCU is expected to mark the sensors as failed. If this occurs, we have proven that sensor

[6] It is not fulfilled, for example, if signal value v_0 is a low bit in a 32-bit command word w and v_1 is a high bit in w, so that, after illegally masking the higher bits of w, v_0 is still correctly visible, but v_1 has been illegally set to 0.

state changes of SD5,6 are not lost within the SDCU. Since the SUT implementation really shows the expected messages, a signal deletion error can be excluded.

Test: Signal-deletion

In this test, the sensors SD5,6 constantly send the state failure to the SDCU. The internal state of the sensors is not modified, just the value in the request frame is adjusted. As it is recorded at TM 00029031597 μs, the sensors are marked failed. Therefore the occurrence of a signal deletion fault can be excluded.

```
TM 00029031597 AM   4 0 : ( 27 ) SDCU:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  0  0  1  1  0  0
                        :   FAILURE: 0  0  0  0  1  1  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  0  0  0  1  1
                        :      COMP: 1  2  3  4
                        :     ALARM: 1  0  1  0
```

4.4.3 Falsification tests for path delay errors.

A path delay error occurs when the specified transition is not completely lost, but occurs too late and/or needs an additional stimulus to be triggered. This type of error would be revealed by waiting for a sufficiently long time for the expected output to occur (of course, an upper bound must be known to perform this test), and toggling the inputs in a way that should stimulate *a sequence of output value changes* instead of only one change. In our case, the additional test would consist of exercising an input sequence on the Comp2 temperature and smoke status which would result in a sequence of ALARM → STANDBY → ALARM → STANDBY → . . . state changes of sensors SD5,6.

Test: Path-delay

In this test, the environment for compartment Comp2 is modified in a way that the sensors will send a sequence of ALARM → STANDBY → ALARM → STANDBY → . . . to the SDCU. As can be seen from the test log, the environment alternates the values for smoke and temperature and the corresponding sensors change their state accordingly. The environment is set to "no fire", so are the sensors SD5,6:

```
TM 00010026978 AM   4 0 : ( 18 ) ENVIRONMENT:
                        : ---------------------------
                        :     COMP:   1    2    3    4
                        : HUMIDITY:   5    5    5    5
                        :    SMOKE:  90    2   90    2
                        :     TEMP:  30   20   30   20

TM 00010027377 AM   4 0 : ( 18 ) SENSORS SDS:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
```

```
                        :     ALARM: 1  1  1  1  0  0  1  1  0  0
                        :    DEFECT: 0  0  0  0  0  0  0  0  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  1  1  0  0  1  1

TM 00010027856 AM   4 0 : ( 18 ) SDCU:
                        : --------------------
                        :     COMP: 1  2  3  4
                        :    ALARM: 1  0  1  0
```

In the next test step, the environment is set to "fire" and the sensors change their states accordingly:

```
TM 00010526998 AM   4 0 : ( 19 ) ENVIRONMENT:
                        : ----------------------------
                        :     COMP:   1    2    3    4
                        : HUMIDITY:   5    5    5    5
                        :    SMOKE:  90   90   90    2
                        :     TEMP:  30   30   30   20

TM 00010527398 AM   4 0 : ( 19 ) SENSORS SDS:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  1  1  1  1  0  0
                        :    DEFECT: 0  0  0  0  0  0  0  0  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  0  0  0  1  1

TM 00010527908 AM   4 0 : ( 19 ) SDCU:
                        : --------------------
                        :     COMP: 1  2  3  4
                        :    ALARM: 1  0  1  0
```

Again, the environment is set to "no alarm". The sensors are set to state standby.

```
TM 00011027004 AM   4 0 : ( 20 ) ENVIRONMENT:
                        : ----------------------------
                        :     COMP:   1    2    3    4
                        : HUMIDITY:   5    5    5    5
                        :    SMOKE:  90    2   90    2
                        :     TEMP:  30   20   30   20

TM 00011027416 AM   4 0 : ( 20 ) SENSORS SDS:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  0  0  1  1  0  0
                        :    DEFECT: 0  0  0  0  0  0  0  0  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  1  1  0  0  1  1
```

```
TM 00011027874 AM   4 0 : ( 20 ) SDCU:
                        : --------------------
                        :     COMP: 1  2  3  4
                        :    ALARM: 1  0  1  0
```

Also it can be noticed that the SDCU is not changing the state for Comp2. Therefore the occurrence of a path delay error can be excluded.

4.4.4 Falsification tests for missing transition errors.
An omitted transition means that the SUT illegally remains in a given state, though a new trigger event occurred. This situation can be uncovered by black-box tests, using the *characterisation traces* of the state machine defining the transition (see [4] for details): The test execution is extended by additional input sequences, so that the associated SUT reactions reveal whether the required transition has been performed or not.

For our system, a command to dump the internal BITE (Built-In Test Equipment) memory can be used as such an additional input sequence [7]: Every alarm is not only indicated on the `compAlarm[c]` interfaces, but also recorded in the internal BITE memory whose contents can be accessed using commands sent to the SDCU via AFDX.

4.4.5 Falsification tests for Stuck-at-0/1 errors.
For suspected stuck-at-0/1 errors [8], a test is designed in such a way that a change in the output interface y where the failure was observed only depends on a single input x. Then x is changed in such way that it should trigger corresponding y changes. If these occur, a "stuck-at situation" cannot be present.

Test: Stuck-at-zero

In our case, we perform a test where SD5 stays continuously in state ALARM, while the state of SD6 is toggled between STANDBY and ALARM. Then, in a correctly operating SUT, an output `compAlarm[2]=1` should occur if and only if SD6 is in state ALARM. In the following test log, it can be seen that no compartment alarm for Comp2 is raised because only sensor SD5 is in state ALARM whereas sensor SD6 remains in state STANDBY. This is the expected behaviour.

```
TM 00021029341 AM   4 0 : ( 20 ) SDCU:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  1  0  1  1  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  1  0  0  1  1
                        :      COMP: 1  2  3  4
                        :     ALARM: 1  0  1  0
```

[7] Not shown here due to space constraints.
[8] For integrated circuits this is discussed in [10, pp. 335ff]

```
TM 00022029307 AM   4 0 : ( 21 ) SDCU:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  1  0  1  1  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  1  0  0  1  1
                        :      COMP: 1  2  3  4
                        :     ALARM: 1  0  1  0
```

In the next steps, both sensors are in state ALARM, which can be seen in the test execution log. But still no alarm for compartment Comp2 is signalled.

```
TM 00023029318 AM   4 0 : ( 22 ) SDCU:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  1  1  1  1  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  0  0  0  1  1
                        :      COMP: 1  2  3  4
                        :     ALARM: 1  0  1  0

TM 00024029314 AM   4 0 : ( 23 ) SDCU:
                        : ---------------------------------------
                        : SENSOR ID: 1  2  3  4  5  6  7  8  9 10
                        :     ALARM: 1  1  1  1  1  1  1  1  0  0
                        :   FAILURE: 0  0  0  0  0  0  0  0  0  0
                        :   STANDBY: 0  0  0  0  0  0  0  0  1  1
                        :      COMP: 1  2  3  4
                        :     ALARM: 1  0  1  0
```

As a consequence, this test could not falsify the stuck-at-0 fault.

4.4.6 Repeated Application of Steps 3 and 4.

If all fault hypotheses in the fault tree elaborated in Step 3 have been falsified, but the tree has not been completely refined, then we return to Step 3 to further extend selected leaves of the fault-tree. After that, Step 4 is repeated.

If the tree was completely refined and no error was found during the execution of these steps, there is a good chance that one of the additional tests failed to cover the problem. In this case there are only two ways of detecting the error: either to start at the end of the trees and do "backtracking" or to start at the beginning, performing validations for each additional test. The choice of further actions then depends on the specific problem and will not be elaborated here.

4.5 Step 5: Error Identification

In our case, all fault hypotheses but the potential stuck-at-0 fault can be falsified by means of additional tests. The internal structure of the SDCU must be taken into account, so that the global input variables `sensorSDSAlarm[]` of the

`processMessages()` method become visible. An additional unit test of this method reveals the presence of a stuck-at-0 fault within this method.

In order to aid with error diagnosis on source-code level a software tool has been developed: Using the interval analysis techniques described in [7, pp. 9ff] (based on [9]), the Interactive Interval Analyser can easily help to identify the error. The Interval Analyser works on compiler-generated control flow graph (CFG) information. The user can select a CFG (which corresponds to a function) from the compilation unit. Subsequently a window containing the function's inputs and outputs is displayed: The inputs are global variables as well as function parameters, while the outputs are again global variables and the function's return value.

The user may now change the intervals assigned to the inputs and the Analyser will interactively show the impact of the changes on the outputs (Fig. 5).

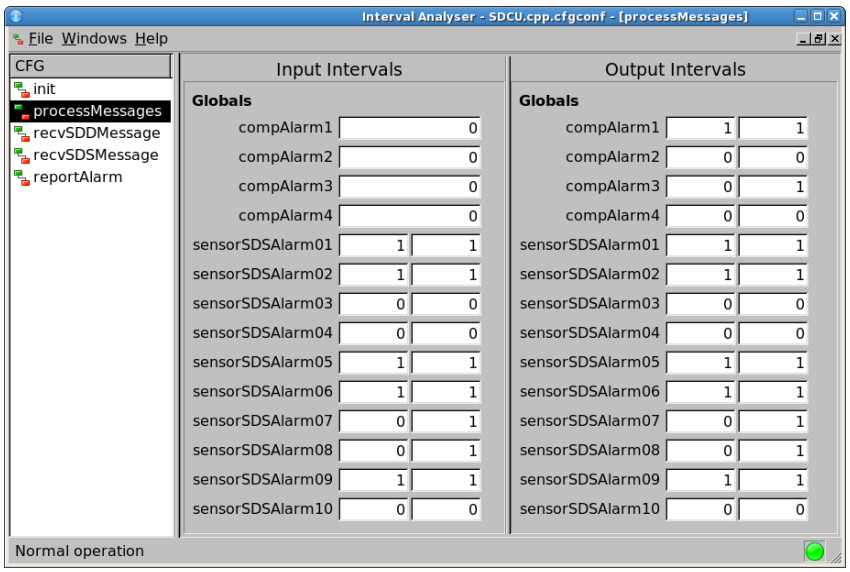

Fig. 5. Interval Analyser

The Analyser can also be used in combination with Verified's *RT-Tester User Interface* (RTTUI). It is then possible to observe the code coverage achieved with the current input intervals. The RTTUI will display the CFG file [9] and colourise all blocks that have been reached during the last execution. This feature can be used for error diagnosis: Setting both inputs `sensorSDSAlarm[5]` and `sensorSDSAlarm[6]` to the interval $[1, 1]$ does not lead to coverage of block 8 (where `compAlarm[2]` would have been set to 1), as shown in Fig. 6.

The last step is to match the corresponding lines of code to the block 8 of the CFG. This leads to the position where the error is located within the source code:

```
if ( sensorSDSAlarm[5] && sensorSDSAlarm[13] ) {
    compAlarm[2] = 1;
}
```

This reveals the observed error: for compartment Comp2, the alarm decision depends on $sensorSDSAlarm[5] \wedge sensorSDSAlarm[13]$ instead of

[9] Displaying the original C/C++ source code is currently not possible. This feature is planned for a later version.

$sensorSDSAlarm[5] \wedge sensorSDSAlarm[6]$.

5 Conclusion

Our experience has shown that it is possible to locate any kind of fault when following the presented procedures (and refinements). Being state of the art for semiconductor error diagnostics, to our knowledge the presented strategy for fault detection is not yet commonly used within the area of distributed embedded systems. Through the tight integration of the procedures with the tool chain a reliable and efficient way of testing is introduced. The tool chain, consisting of the RT-Tester, its user interface (RTTUI) and the Interval Analyser, can be supplemented by tools [10] like Relex FTA or Isograph FaultTree+ for fault tree analysis.

References

[1] Abraham, J. A. and W. K. Fuchs, *Fault and error models for VLSI*, IEEE Proceedings **74** (1986), pp. 639–654.

[2] Badban, B., M. Fränzle, J. Peleska and T. Teige, *Test automation for hybrid systems*, 2006, submitted to Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006), Available under `http://www.informatik.uni-bremen.de/agbs/projects/hybris/peleska_et_al_soqua2006.pdf`.

[3] Binder, R. V., "Testing Object-Oriented Systems: Models, Patterns, and Tools," Addison-Wesley, 2000.

[4] Chow, T. S., *Testing software design modeled by finite-state machines*, IEEE Transactions on Software Engineering **SE-4** (1978), pp. 178–186.

[5] *Software Considerations in Airbone Systems and Equipment Certification*, RTCA/DO-178B (1992).

[10] No tool support was used for the fault trees in this paper.

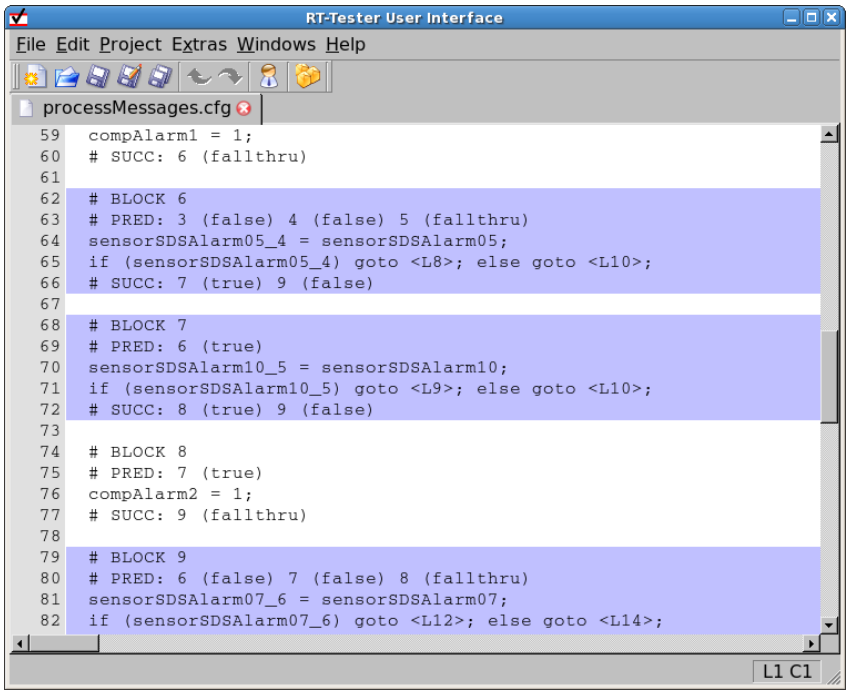

Fig. 6. Covered CFG blocks shown in RT-Tester UI.

[6] Eggersglüß, S., G. Fey and R. Drechsler, *SAT-based ATPG for Path Delay Faults in Sequential Circuits*, in: *IEEE International Symposium on Circuits and Systems (ISCAS 2007)*, New Orleans, 2007, pp. 3671–3674.

[7] Hannemann, U., T. Hartmann, J. Peleska and A. Tsiolakis, *Analysis of Interfaces and Signal Flow in the Cabin Domain*, Technical report, University of Bremen, TZI (2006), deliverable for KATO-TP13.

[8] International Electrotechnical Commission, Geneva, "International Standard IEC1025 - Fault Tree Analysis (FTA)," (1990).

[9] Jaulin, L., M. Kieffer, O. Didrit and É. Walter, "Applied Interval Analysis," Springer-Verlag, London, 2001.

[10] Miczo, A., "Digital Logic Testing and Simulation," Wiley-Interscience, 2003.

[11] Storey, N., "Safety-Critical Computer Systems," Addison Wesley Longman, 1996.

[12] Verified Systems, *RT-Tester 6.x – User Manual*, Technical Report Verified-INT-014-2003, Verified Systems International GmbH, Bremen (2004).

Symbolic and Abstract Interpretation for C/C++ Programs

Helge Löding[2,4]

GESy Graduate School of Embedded Systems
University of Bremen
and Verified Systems International GmbH
Germany

Jan Peleska[1,3]

Centre of Information Technology
University of Bremen
Germany

Abstract

We present a construction technique for abstract interpretations which is generic in the choice of data abstractions. The technique is specialised on C/C++ code, internally represented by the GIMPLE control flow graph as generated by the gcc compiler. The generic interpreter handles program transitions in a symbolic way, while recording a history of symbolic memory valuations. An abstract interpreter is instantiated by selecting appropriate lattices for the data types under consideration. This selection induces an instance of the generic transition relation. All resulting abstract interpretations can handle pointer arithmetic, type casts, unions and the aliasing problems involved. It is illustrated how switching between abstractions can improve the efficiency of the verification process. The concepts described in this paper are implemented in the test automation and static analysis tool RT-Tester which is used for the verification of embedded systems in the fields of avionics, railways and automotive control.

Keywords: automated testing, static analysis, abstract interpretation, Galois connections

1 Introduction

1.1 Objectives and Overview

Concrete and abstract interpretation are core mechanisms for automated static analysis, test case/test data generation and property checking of software: The concrete interpretation helps to explore program (component) behaviour with concrete data values without having to compile, link and execute the program on the

[1] Email: jp@tzi.de
[2] Email: hloeding@tzi.de
[3] Partially supported by the BIG Bremer Investitions-Gesellschaft under research grant 2INNO1015B
[4] Supported by a research grant of the Graduate School in Embedded Systems GESy http://www.gesy.info

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

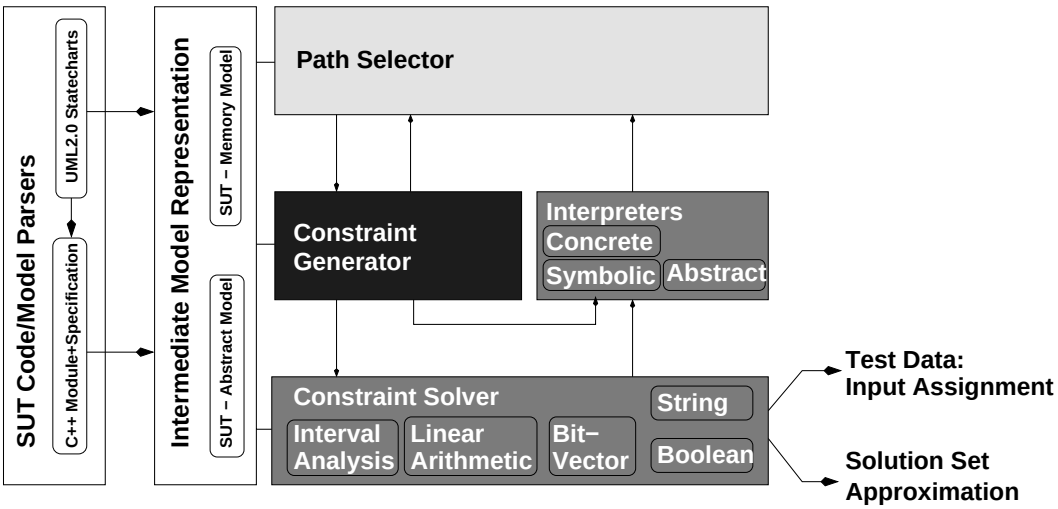

Fig. 1. Building blocks of tools for test automation, static analysis and property verification.

target platform. The abstract interpretation reduces the complexity of verification goals or, more general, reachability problems, by abstracting from details which are unnecessary for the goal under consideration.

Consider the building blocks typically present in tools supporting test automation, static analysis and/or property checking as shown in Fig. 1: The program code to be analysed or a specification model are transformed into a uniform *intermediate model representation (IMR)* which is independent of the concrete SUT code or specification syntax. This reduces the dependencies between concrete syntax and analysis algorithms. Most of the problems arising in automated test case/test data generation, static analysis and property verification can be paraphrased as reachability problems, as has been pointed out in [10]. Therefore a *path selector* performs a choice of potential paths through the model to be checked with respect to feasibility: The goal is solved if concrete input data can be found so that the software component under analysis executes along one of the suggested paths. While the general reachability problem is undecidable, concrete goals can often be realised in a highly efficient way. To this end, the *constraint generator* constructs a collection of constraints to be met in order to provoke an execution along the selected paths. The construction requires a *symbolic interpreter*, a tool component for collecting the guard conditions along the selected paths. With a sufficient collection of constraints at hand, the *constraint solver* tries to construct concrete data solving the constraints or to prove their infeasibility.

The choice of the *abstract* interpretation technique considerably influences the efficiency of automated solvers used for these purposes: For proving that a constraint collection can never be satisfied it is often more efficient to show this for an abstracted program version, so that this also implies infeasibility for the concrete program. Conversely, some abstractions are especially useful for under-approximating the solution set of the constraints given, so that any data vector of this approximation represents a solution.

In this paper we focus on interpreters for C/C++ programs. For this task it is necessary to capture all "side effects" of aliasing, pointer arithmetic, type casts

and unions possibly occurring in C/C++ software, so that no hidden effects of instructions on the valuation of symbols not occurring in the statement are missed during the interpretation process. We first present operational rules for a concrete semantics covering these aspects (Section 3). Next we observe that for a given collection of constraints, the efficiency of the solver strongly depends on the choice of abstraction. As a consequence it is desirable to switch abstractions for one and the same data type during the interpretation while still ensuring the correctness of the interpretation results. This objective is met by means of a *symbolic* interpreter for C/C++ programs (Section 4): This tool component handles program transitions in a symbolic way, while recording a history of symbolic memory valuations. The valuations are represented by memory addresses (these are necessary in order to cope with the aliasing problems), value expressions and application conditions: A memory item is only valid if a valuation of inputs can be found so that the application condition becomes true. Finally we describe how abstract interpreters can be constructed by instantiating the symbolic interpreter with lattices to be used for abstracting the data types involved (Section 5). As a consequence, the basic interpretation algorithm can be completely re-used for each choice of abstraction lattice, only functions for the valuation of expressions in the context of the selected lattices have to be added. In Section 6 an example is given which illustrates the mechanics and the effects of symbolic and abstract interpretation.

1.2 Background and Related Work

The full consideration of C/C++ aliasing situations with pointers, casts and unions is achieved at the price of lesser performance. In [4,2], for example, it is pointed out how more restrictive programming styles, particularly the avoidance of pointer arithmetics, can result in highly effective static analyses with very low rates of false alarms. Conversely it is pointed out in [14] that efficient checks of pointer arithmetics can be realised if only some aspects of correctness (absence of out-of-bounds array access) are investigated. As another alternative, efficient static analysis results for large general C-programs can be achieved if a higher number of false alarms (or alternatively, a suppression of potential failures) is acceptable [5], so that paths leading to potential failures can be identified more often on a syntactic basis without having to fall back on constraint solving methods.

On the level of binary program code verification impressive results have been achieved for certain real-world controller platforms, using explicit representation models [12]. These are, however, not transferable to the framework underlying our work, since the necessity to handle floating point and wide integer types (64 or 128 bit) forbids the explicit enumeration of potential input values and program variable states.

All techniques described in this paper are implemented in the RT-Tester tool developed by the authors and their research group at the University of Bremen in cooperation with Verified Systems International GmbH [15]. In [10] we have motivated in more detail why testing, static analysis and property checking of software code should be considered as an integrated verification task, so integrated tool support for these complementary aspects of software verification is highly desirable. The approach pursued with the RT-Tester tool differs from the strategies of other

authors [4,2,14]: We advocate an approach where test and verification activities focus on small program units (a few functions or methods) and should be guided by the expertise of the development or verification specialists. Therefore the RT-Tester tool provides mechanisms for specifying preconditions about the expected or admissible input data for the unit under inspection as well as for semi-automated stub ("mock-object") generation showing user-defined behaviour whenever invoked by the unit to be analysed. As a consequence, programmed units can be verified immediately and interactive support for bug-localisation and further investigation of potential failures is provided. The SMT constraint solver used in the tool is based on ideas described in [11,1,6].

2 Theoretical Foundations

Recall that a binary relation $\sqsubseteq$ on a set L is called a *(partial) order* if $\sqsubseteq$ is reflexive, transitive and anti-symmetric. An element $y \in L$ is called an *upper bound of* $X \subseteq L$ if $x \sqsubseteq y$ holds for all $x \in X$. The lower bound of a set is defined dually. An upper bound y' of X is called a *least upper bound of* X and denoted by $\sqcup X$ if $y' \sqsubseteq y$ holds for all upper bounds y of X. Dually, the *greatest lower bound* $\sqcap X$ of a set X is defined.

An ordered set $(L, \sqsubseteq)$ is called a *complete lattice*, if $\sqcap X$ and $\sqcup X$ exist for all subsets $X \subseteq L$. Lattice L has a *largest element* (or *top*) denoted by $\top =_{\text{def}} \sqcup L$ and a *smallest element* (or *bottom*) denoted by $\bot =_{\text{def}} \sqcap L$. Least upper bounds and greatest lower bounds induce binary operations $\sqcup, \sqcap : L \times L \to L$ by defining $x \sqcup y =_{\text{def}} \sqcup\{x, y\}$ (the *join* of x and y) and $x \sqcap y =_{\text{def}} \sqcap\{x, y\}$ (the *meet* of x and y), respectively. If the join and meet are well-defined for an ordered set $(L, \sqsubseteq)$ but $\sqcup X, \sqcap X$ do not exist for all $X \subseteq L$ then $(L, \sqsubseteq)$ is called an *(incomplete) lattice*.

From the collection of canonic ways to construct new lattices from existing ones $(L, \sqsubseteq), (L_1, \sqsubseteq_1), (L_2, \sqsubseteq_2)$, we need (1) cross products $(L_1 \times L_2, \sqsubseteq')$ where the partial order is defined by $(x_1, x_2) \sqsubseteq' (y_1, y_2)$ if and only if $x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$ and (2) partial function spaces $(V \nrightarrow L, \sqsubseteq')$ where $f \sqsubseteq' g$ for $f, g \in V \nrightarrow L$ if and only if $\text{dom } f \subseteq \text{dom } g \wedge (\forall x \in \text{dom } f : f(x) \sqsubseteq g(x))$.

Mappings $\phi : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ between ordered sets are called *monotone* if $x \sqsubseteq_1 y$ implies $\phi(x) \sqsubseteq_2 \phi(y)$ for all $x, y \in L$. Mappings $\phi : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ between lattices are called *homomorphisms* if they respect meets and joins, that is, $\phi(x \sqcup_1 y) = \phi(x) \sqcup_2 \phi(y)$ and $\phi(x \sqcap_1 y) = \phi(x) \sqcap_2 \phi(y)$ for all $x, y \in (L_1, \sqsubseteq_1)$. Since $x \sqsubseteq_1 y$ implies $x \sqcup_1 y = y$ and $x \sqcap_1 y = x$, homomorphisms are monotone.

A *Galois connection (GC)* between lattices $(L_1, \sqsubseteq_1), (L_2, \sqsubseteq_2)$ is a tuple of mappings $_^{\triangleright} : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ (called *right*) and $_^{\triangleleft} : (L_2, \sqsubseteq_2) \to (L_1, \sqsubseteq_1)$ (called *left*) such that $a^{\triangleright} \sqsubseteq_2 b \Leftrightarrow a \sqsubseteq_1 b^{\triangleleft}$ for all $a \in L_1, b \in L_2$. This defining property implies that Galois connections are monotone in both directions.

Given any transition system $TS = (S, S_0, \longrightarrow)$ with state space S, initial states in $S_0 \subseteq S$ and transition relation $\longrightarrow \subseteq S \times S$, the most fine-grained state space abstraction possible is represented by the power set lattice $L_{\mathbf{P}}(S) = (\mathbf{P}(S), \subseteq)$ with join operation $\cup$ and meet $\cap$. We introduce an abstract interpretation semantics on $L_{\mathbf{P}}(S)$ by turning it into a state transition system $TS_{\mathbf{P}} = (L_{\mathbf{P}}(S), \{S_0\}, \longrightarrow_{\mathbf{P}})$ by lifting the original transition relation to sets: Using Plotkin-style notation, this can

be specified as

$$\frac{\forall i \in I, s_i, s_i' \in S : s_i \longrightarrow s_i'}{\{s_i \mid i \in I\} \longrightarrow_{\mathbf{P}} \{s_i' \mid i \in I\}}$$

Compared to the original transition system TS, this abstract interpretation $\longrightarrow_{\mathbf{P}}$ introduces no loss of information, since its restriction to pairs of singleton sets is equivalent to the original transition relation:

$$\forall s_1, s_2 \in S : s_1 \longrightarrow s_2 \Leftrightarrow \{s_1\} \longrightarrow_{\mathbf{P}} \{s_2\}$$

It is, however, an abstraction, since for transitions between states with cardinality higher than one, say $\{s_1, s_2, \ldots\} \longrightarrow_{\mathbf{P}} \{s_1', s_2', \ldots\}$, only the possible resulting states are listed $(s_1', s_2', \ldots)$ but the information whether, for example, $s_1 \longrightarrow s_1'$ or $s_1 \longrightarrow s_2'$ is no longer available.

Now, given any other transition system $TS_L = (L, L_0, \longrightarrow_L)$ based on a lattice $(L, \sqsubseteq)$ we can check whether TS_L is a valid abstract interpretation of TS by the aid of $TS_{\mathbf{P}}$ and Galois connections:

Definition 2.1 Transition system $TS_L = (L, L_0, \longrightarrow_L)$, based on a lattice $(L, \sqsubseteq)$, is a *valid abstract interpretation of* $TS = (S, S_0, \longrightarrow)$ if (i) there exists a Galois connection $(\mathbf{P}(S), \subseteq) \overset{\triangleleft}{\underset{\triangleright}{\leftrightarrows}} (L, \sqsubseteq)$, (ii) the transition relation $\longrightarrow_L$ is a *valid abstract relation* the sense that $\forall a, a', b \in L : (a \longrightarrow_L a' \wedge b \sqsubseteq a \Rightarrow \exists b' \in L : b \longrightarrow_L b' \wedge b' \sqsubseteq a')$, (iii) the transition relation $\longrightarrow_L$ satisfies $\forall (p, p') \in \longrightarrow_{\mathbf{P}} : \exists a' \in L : p^{\triangleright} \longrightarrow_L a' \wedge p'^{\triangleright} \sqsubseteq a'$ and (iv) the transition relation $\longrightarrow_L$ satisfies $\forall (a, a') \in \longrightarrow_L : \exists p' \in \mathbf{P}(S) : a^{\triangleleft} \longrightarrow_{\mathbf{P}} p' \wedge p' \subseteq a'^{\triangleleft}$.

The following theorem provides a "recipe" for constructing valid abstract interpretations, as soon as a GC according to Definition 2.1, (i) has been established:

Theorem 2.2 *Given lattice* $(L, \sqsubseteq)$ *and Galois connection* $(\mathbf{P}(S), \subseteq) \overset{\triangleleft}{\underset{\triangleright}{\leftrightarrows}} (L, \sqsubseteq)$*, define transition system* $TS_L = (L, L_0, \longrightarrow_L)$ *by (i)* $L_0 = \{S_0\}^{\triangleright}$*, (ii)* $\frac{p^{\triangleright\triangleleft} \longrightarrow_{\mathbf{P}} p'}{p^{\triangleright} \longrightarrow_L p'^{\triangleright}}$ *and (iii)* $\frac{a^{\triangleleft} \longrightarrow_{\mathbf{P}} p'}{a \longrightarrow_L p'^{\triangleright}}$ *Then* TS_L *is a valid abstract interpretation of* TS *in the sense of Definition 2.1.*

For more details about lattices and GC and the proof of Theorem 2.2 the reader is referred to [3,9].

3 Control Flow Graphs and GIMPLE, Concrete Semantics

3.1 GIMPLE Programs

We use the gcc compiler to transform a given C/C++ program into GIMPLE code. As described in [7,8], this semantically equivalent representation of a program constitutes an intermediate transformation result from source to assembler, where all expressions appearing in statements contain at most one operator and (with the exception of function invocations) at most two operands. Operands may only be variable names or nested structure and array accesses (henceforth called selectors)

as well as constant values. By introducing auxiliary variables, all original statements will be transformed to adhere to this requirement. Statements may therefore only be assignments from expressions to variables (or atomic selectors in the above sense). Casting and referencing/dereferencing of variables (or selectors) form expressions in themselves, and may therefore not be used as operands, but instead need to be executed as separate assignments to auxiliary variables. GIMPLE programs contain no loop constructs. Instead, all loops from the original source are transformed into conditional jumps to preceeding labelled statements. GIMPLE therefore contains only two different types of branching statements:

```
<if-else-stmt> ::= if ( <condition> ) goto <label>; else goto <label>;
<switch-stmt>  ::= switch ( <variable> ) { <cases>  <default>_opt }
<cases>        ::= <cases>_opt case <value>: goto <label>;
<default>      ::= default: goto <label>;
```

For the description of concrete GIMPLE semantics we encode each GIMPLE function as a *control flow graphs (CFGs)*. Each function/method of a C/C++ program is associated with a CFG. Each CFG G has a distinguished initial node $I(G)$ corresponding to function entry and a terminal node $O(G)$ corresponding to function return. Each CFG node is labelled with a single GIMPLE statement, each edge with a GIMPLE branching condition. For sequences of non-branching statements, the edges are labelled with `true`. Branching statements are represented as edges labelled with the applicable branching conditions, each edge pointing to the target node referenced in the `goto <label>` statement in the GIMPLE code.

The concrete operational semantics of a GIMPLE program P, represented by a collection of control flow graphs as described above, will now be explained by associating a transition system with P.

3.2 GIMPLE state space

For representing the semantics of GIMPLE programs P, we use the following class of transition systems $TS_G = (S_G, S_0, \longrightarrow_G)$. The program state space is defined as

$$S_G = N(P) \times (Seg \times \mathbb{N}_0 \nrightarrow Symbols) \times (Seg \times \mathbb{N}_0 \nrightarrow BYTE^*)$$

with typical element $(n, \nu, \mu) \in S_G$. Set $N(P)$ comprises all nodes in the CFGs associated with any function of P. The second and third component of this Cartesian product represent function spaces for address mappings and memory state: For modelling the association between variables, their aliases and their associated memory portions, we introduce (1) a partial function $\nu : Seg \times \mathbb{N}_0 \nrightarrow Symbols$ mapping existing virtual addresses on the segment of type $Seg = \{stack, heap, global, code\}$ to a symbol (variable or function) associated with this address and (2) a partial function $\mu : Seg \times \mathbb{N}_0 \nrightarrow BYTE^*$ associating with each existing virtual address a sequence of bytes, representing the current memory valuation of the given address.

The set $Symbols$ only contains the *basic* symbol names, that is, the name `a` of an array, but not the array element `a[4]` and the name of a structured variable `x` but not the name of `x.y.z[5]` of a structure component. Component and array element identifiers are called *selectors* and comprised in a (possibly infinite) set $Selectors$ which is a superset of $Symbols$, since each basic name is a selector, too.

The initial state of S_G is $S_0 = \{(I(f), \nu_0, \mu_0)\}$, where $I(f)$ is the initial node of the CFG associated with the GIMPLE function of interest, ν_0 contains all addresses of global variables and actual parameters used in the invocation of $f()$ and μ_0

contains the memory portions associated with these actual parameters and of all global variables, initialised according to the precondition on which the execution of $f()$ should be based.

3.3 Auxiliary functions

For recording state changes in S_G and determining the current state of variable valuations some auxiliary functions are needed.

Given an arbitrary selector, function $\beta : \mathit{Selectors} \to \mathit{Symbols}$ returns its base symbol, e.g. for $\beta(\texttt{x.y.z[5]}) = x$. This will be required to retrieve base addresses for selectors by means of ν.

Since virtual addresses are unique across memory segments, a function $\hat{\nu} : \mathit{Symbols} \not\to \mathbb{N}_0$ mapping identifiers to their respective address is well-defined when taking scoping into account. For a given symbol that is defined both within the stack and global segments, $\hat{\nu}$ will return the virtual address corresponding to the symbol definition within the stack.

$\hat{\nu}$ can be extended to map from selectors to virtual base addresses to yield $\nu^- : \mathit{Selectors} \not\to \mathbb{N}_0$ with $\nu^-(\mathit{sel}) =_{\text{def}} \hat{\nu}(\beta(\mathit{sel}))$.

Given an arbitrary selector, function $\omega : \mathit{Selectors} \to \mathbb{N}_0$ returns the *bit offset* of the selector's memory location from its base address. The offset is measured in bits so that also operations on bitfields can be captured. This information is obviously platform-specific: ω is constructed from the size and alignment information provided by the gcc compiler on the specific platform it is used. As with ν^-, the appropriate memory segment for multiply defined base symbols is determined by first assessing symbol definitions within the stack segment.

Function $\tau : \mathit{Selectors} \not\to \mathit{Types}$ returns the type for any given selector. The type information is then gained from the internal type data gathered by the gcc compiler. Again, scoping is taken into account.

Function τ may be extended to determine the type of a given expression forming $\tau^* : \mathit{Expr} \not\to \mathit{Types}$ by taking (return) types of used operands and operators into account. If a given selector corresponds to a pointer type, then function $\vec{\tau} : \mathit{Selectors} \not\to \mathit{Types}$ may be used to obtain its target type.

Function $\sigma : \mathit{Types} \to \mathbb{N}_0$ is used to determine a given type's size in bits.

The state space only records the current memory state as sequences of bytes. Function $\iota : \mathit{BYTE}^* \times \mathit{Types} \not\to D$ is used to interpret a given sequence of bytes as a specific type. Here, D denotes the union of all atomic domains. It is only defined for byte sequences long enough to hold a value of given type. Conversely, we define $\iota^- : D \times \mathit{Types} \not\to \mathit{BYTE}^*$ to be the byte representation for a given value with known type. For these functions, the size of atomic types, encoding methods and the little or big endianess of the platform has to be determined. This information is retrieved from the gcc type- and debugging information.

For reading data from memory, we initially define $\epsilon^a : S_G \times \mathbb{N}_0 \times \mathbb{N}_0 \not\to \mathit{BYTE}^*$. Function application $\epsilon^a((\nu, \mu), a, s)$ reads a bit sequence of a given length s beginning from a specified address a within the memory, and returns its contents as byte sequence. For this, we find the segment and base address $(\mathit{seg}, a_{\mathit{base}})$ within $\mathit{dom}(\nu)$, for which byte sequence $\mu(\mathit{seg}, a_{\mathit{base}})$ encloses address a. If specified size s exceeds

byte sequence $\mu(seg, a_{base})$ beginning from a, ϵ^a has to take direct successor byte sequences within seg into account to be defined. If size s is not a multiple of 8, the resulting byte sequence will be constructed by adding additional high order 0 bits until its bitsize reaches the next higher multiple of 8.

Using ϵ^a, we now construct a function to read raw byte data from memory using selectors. We define $\epsilon^s : S_G \times Selectors : BYTE^*$ as

$$\epsilon^s((\nu,\mu), sel) =_{\text{def}} \epsilon^a((\nu,\mu), \nu^-(sel) + \omega(sel), \sigma(\tau(sel)))$$

We now define a function $\epsilon^e : S_G \times Expr \not\to BYTE^*$, which evaluates a given GIMPLE expression according to the current memory valuation. As GIMPLE expressions contain at most one operator, we can do this by distinguishing different expression types. For expressions consisting of constant values or selectors, ϵ^e corresponds to applications of ι^- or ϵ^s respectively. Other types of expressions may be evaluated using one of the following definitions of ϵ^e:

Let $\square \in \{+, -, *, /, \%, \wedge, \vee, >, <, \geq, \leq, =, \neq\}$ be a binary arithmetic or boolean operator, and let $exp =_{\text{def}} exp_1 \square exp_2$ be an application to two operand expressions. We define

$$\begin{aligned}&\epsilon^e((\nu,\mu), exp) =_{\text{def}}\\ &\quad \iota^-(\iota(\epsilon^e((\nu,\mu), exp_1), \tau^*(exp_1)) \square \iota(\epsilon^e((\nu,\mu), exp_2), \tau^*(exp_2)), \tau^*(exp))\end{aligned}$$

Concurrently, for an unary arithmetic or boolean operator $\diamond \in \{+, -, !\}$ and expression $exp =_{\text{def}} \diamond exp_1$ we define

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \iota^-(\diamond\, \iota(\epsilon^e((\nu,\mu), exp_1), \tau^*(exp_1)), \tau^*(exp))$$

For a bitwise operator $\circ \in \{\&, |, \mathbf{XOR}\}$ and expression $exp =_{\text{def}} exp_1 \circ exp_2$, the operation is performed on raw byte data, and we define

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \epsilon^e((\nu,\mu), exp_1) \circ \epsilon^e((\nu,\mu), exp_2)$$

For bitwise unary operator $\sim$ and according expression $exp =_{\text{def}} \sim exp_1$, we define

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \sim \epsilon^e((\nu,\mu), exp_1)$$

For a shift operator $\triangle \in \{\ll, \gg\}$ and expression $exp =_{\text{def}} exp_1 \triangle exp_2$, the operation is performed as follows (note that exp_2 must correspond to an integral type):

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \epsilon^e((\nu,\mu), exp_1) \triangle \iota(\epsilon^e((\nu,\mu), exp_2), \tau(exp_2))$$

Dereferencing of a selector, $exp =_{\text{def}} *sel$, may be evaluated using

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \epsilon^a((\nu,\mu), \iota(\epsilon^s((\nu,\mu), sel), \mathbb{N}_0), \sigma(\vec{\tau}(sel)))$$

Conversely, referencing of a selector, $exp =_{\text{def}} \&sel$, is defined as

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \iota^-(\nu^-(sel) + \omega(sel), \mathbb{N}_0)$$

For a cast expression $exp =_{\text{def}} (t)(exp_1)$ with target type t, we define

$$\epsilon^e((\nu,\mu), exp) =_{\text{def}} \iota^-((t)_C\, \iota(\epsilon^e((\nu,\mu), exp_1), \tau^*(exp_1)), t)$$

where cast operator $()_C$ uses C cast operator semantics for atomic types t and $\tau^*(exp_1)$.

For purposes of legibility, we henceforth denote ϵ^e by ϵ unless noted otherwise. For specifying the effect of write operations on the memory, we use function

$$\phi : (\mathbb{N}_0 \times \mathbb{N}_0) \times S_G \times BYTE^* \not\to (Seg \times \mathbb{N}_0 \not\to BYTE^*)$$

To begin with, function application $\phi((a_{tgt}, o_{tgt}), (\nu,\mu), val_{byte})$ determines the target memory segment for target base address a_{tgt} and offset o_{tgt}. It then returns a new memory valuation μ', which differs from μ only in the new valuation of the target segment, starting at target base address a_{tgt} but unchanged before offset o_{tgt}. Starting at the offset, the memory is changed according to the byte sequence val_{byte}.

3.4 Transition relation: operational rules.

The operational rules specifying the transition relation $\longrightarrow_G \subseteq S_G \times S_G$ on the GIMPLE state space are based on the control flow graph representation of each GIMPLE function. In Plotkin-style notation, each rule is of the form

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2,\ \iota(\epsilon((\nu,\mu),g),int) \neq 0}{(n_1,\nu,\mu) \longrightarrow_G (n_2,\nu',\mu')}$$

Informally speaking, a transition $(n_1,\nu,\mu) \longrightarrow_G (n_2,\nu',\mu')$ is possible if (1) there exists an edge from n_1 to n_2 in the respective CFG, (2) the guard condition g associated with this edge evaluates to `true` (for C-like languages this means that it evaluates to an integral value not equal to zero) in the current valuation (ν,μ). For each type of statement encoded in the nodes n_1 it remains to define the *effect* of this statement on (ν,μ), resulting in the new valuation state (ν',μ'). Below we give some examples of detailed rule specifications.

(1) The effect of a *stack variable definition*, $n_1 =_{\text{def}}$ `typex x;`, is to allocate the required space on stack. The values, however, are still undefined. As a consequence the effect on the memory valuation can be specified by

$$\nu' = \nu \oplus [(stack,a') \mapsto x]$$

where a' is a fresh address not occurring in dom ν (in fact, we use the proper offset of x from the base of the stack frame for building a'). The effect on the memory is

$$\mu' = \mu \oplus [(stack,a') \mapsto \underbrace{\langle ?,\ldots,? \rangle}_{\texttt{sizeof(typex)}}]$$

where "?" denotes that the byte values are still undefined.

(2) The effect of an *assignment to a selector*, $n_1 =_{\text{def}}$ `sel = expr;`, is to change the memory at the base address plus offset, as defined by the selector according to the expression valuation. As the left-hand and right-hand sides of the assignment need not necessarily be typed identically, we first construct the artificial cast expression $expr' = (\tau(sel))(expr)$. As we have now ensured $expr'$ to be of the type corresponding to sel, we go on and assign

$$\nu' = \nu,\ \mu' = \phi((\nu^-(sel),\omega(sel)),(\nu,\mu),\epsilon((\nu,\mu),expr'))$$

(3) The effect of an *assignment to a de-referenced selector*, $n_1 =_{\text{def}}$ `*sel = expr;`, is to change the memory at the address pointed to by `sel` according to the expression valuation and the pointer target type of the selector. We therefore need to calculate the target address a_{trg} of the write operation first. This is done by evaluating $a_{trg} = \iota(\epsilon^s((\nu,\mu),sel),\mathbb{N}_0)$. Again using an artificial cast expression $expr' = (\vec{\tau}(sel))(expr)$, we can now construct a new state space by assigning

$$\nu' = \nu,\ \mu' = \phi((a_{trg},0),(\nu,\mu),\epsilon((\nu,\mu),expr'))$$

(4) The effect of copying memory, $n_1 =_{\text{def}}$ `memcpy(trg,src,s);`, is to copy `s` successive bytes starting with address `src` to the memory indicated by `trg`. This may be accomplished by defining

$$a_{src} = \iota(\epsilon((\nu,\mu),src),\mathbb{N}_0)$$
$$a_{trg} = \iota(\epsilon((\nu,\mu),trg),\mathbb{N}_0)$$

to be the addresses specified in `src` and `trg` respectively. We can now construct

$$\mu_0 = \phi((a_{trg},0),(\nu,\mu),\epsilon^a((\nu,\mu),a_{src},8))$$
$$\ldots$$
$$\mu_i = \phi((a_{trg},8*i),(\nu,\mu_{i-1}),\epsilon^a((\nu,\mu_{i-1}),a_{src}+8*i,8))$$
$$\ldots$$
$$\mu_{s-1} = \phi((a_{trg},8*(s-1)),(\nu,\mu_{s-2}),\epsilon^a((\nu,\mu_{s-2}),a_{src}+8*(s-2),8))$$

and finally

$$\nu' = \nu$$
$$\mu' = \mu_{s-1}$$

(5) The effect of a *function invocation*, $n_1 =_{\text{def}} \texttt{sel = f(x1,...,xn);}$, for a function with prototype `t f(t1 z1,...,tk zk)` is calculated according to the following operational rule:

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2,\ \iota(\epsilon((\nu,\mu),g),int) \neq 0,\ (I(G_f),\nu_1,\mu_1) \longrightarrow_G^* (O(G_f),\nu_2,\mu_2)}{(n_1,\nu,\mu) \longrightarrow_G (n_2,\nu',\mu')}$$

In this rule, ν_1, μ_1 are extensions of ν, μ which comprise the initial settings of the formal parameters and the return value:

$$\nu_1 = \nu[(stack,a) \mapsto xReturn, (stack,a_1) \mapsto z_1, \ldots, (stack,a_k) \mapsto z_k]$$

Here $a, a_1, \ldots, a_k$ are fresh address values and $xReturn$ is an auxiliary name for the stack location storing the return value. The initial valuation of $xReturn$ is undefined, but the z_i carry the valuation of their actual parameters x_i:

$$\begin{aligned}
&\mu_1 = \mu_1^1\\
&\mu_1^1 = \phi((\nu_1^-(z_1),0),(\nu_1,\mu_1^2),\epsilon((\nu_1,\mu_1^2),(t_1)(x_1)))\\
&\ldots\\
&\mu_1^i = \phi((\nu_1^-(z_i),0),(\nu_1,\mu_1^{i+1}),\epsilon((\nu_1,\mu_1^{i+1}),(t_i)(x_i)))\\
&\ldots\\
&\mu_1^k = \phi((\nu_1^-(z_k),0),(\nu_1,\mu_1^{k+1}),\epsilon((\nu_1,\mu_1^{k+1}),(t_k)(x_k)))\\
&\mu_1^{k+1} = \mu[(stack,a) \mapsto \underbrace{\langle ?,\ldots,? \rangle}_{\texttt{sizeof(t)}}]
\end{aligned}$$

Now the precondition $(I(G_f),\nu_1,\mu_1) \longrightarrow_G^* (O(G_f),\nu_2,\mu_2)$ in the operational rule above requires that a sequence of transitions through the CFG of f should exist, starting with valuation ν_1, μ_1, so that the final valuation before function return, ν_2, μ_2, defines the target state (n_2, ν', μ') via

$$\mu' = \phi((\nu_2^-(sel),\omega(sel)),(\nu_2,\mu_2),\epsilon((\nu_2,\mu_2),(\tau(sel))(xReturn)))$$

Finally, the local variable addresses and associated memory valuations of f are removed from ν', μ'.

4 Symbolic Interpretation of GIMPLE-Programs

For symbolic interpretation the state space is defined as

$$\begin{aligned}
&S_S = N(P) \times \mathbb{N}_0 \times M\\
&M = dataSegment \times heapSegment \times stackSegment\\
&dataSegment = \textit{M-Item}^*\\
&heapSegment = \textit{M-Item}^*\\
&stackSegment = stackFrame^*\\
&stackFrame = \textit{M-Item}^*\\
&\textit{M-Item} = \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\}) \times BaseAddress \times\\
&\qquad Types \times Offset \times Length \times Value \times Constraint\\
&BaseAddress = String\\
&Offset = Length = Value = Constraint = Expr(Symbols_S)\\
&Symbols_S = Symbols \times \mathbb{N}_0
\end{aligned}$$

Each symbolic state consists of a triple $(node, n, mem)$ where $node$ is a node in the GIMPLE control flow graph representing the current "program counter state" of the symbolic execution, n serves as an instruction counter and mem is the current history state of symbolic memory valuations, called *memory items* $m \in mem$. The collection of memory items generated so far is structured according their allocation

in the data segment, heap or stack, respectively. The stack is further sub-divided into frames, so that the validity of stack variables during their associated function executions can be clearly specified.

The components of a memory item are accessed using $m.v_0$, $m.v_1$, $m.a$, $m.t$, $m.o$, $m.l$, $m.val$, $m.c$ for the respective projections. Component $m.a$ represents the base address of a memory item, typically denoted by $\&x$ if the memory location corresponds to a variable x or by a fictitious address symbol representing the start address of a dynamic memory allocation. Component $m.o$ denotes the offset from the base address, where the value specified in $m.val$ is written to. For writing one value $m.val$, the memory portion starting at $m.a + m.o$ is used, and the length of this portion is determined by the type information $m.t$. If the length specification $m.l$ is a multiple of `sizeof(t)` this specifies that $m.l$/`sizeof(t)` copies of $m.val$ are written into the respective memory segment, starting at $m.a+m.o$. Component $m.c$ represents a symbolic validity constraint for the existence of the item. For concrete or abstract interpretations this means that the memory item is only feasible if $m.c$ – after having been properly resolved – evaluates to true.

For the symbolic specification of offsets, lengths, values and constraints GIMPLE expressions over symbols from $Symbols_S$ are used: Such an expression addresses each identifier as a pair (x, n) where $x \in Symbol$ is an ordinary GIMPLE symbol and n is a version information. Components $m.v_0, m.v_1$ represent validity information: When resolving a symbol $(x, n) \in Symbols_S$ occurring in offset, length, value or constraint expressions of some memory item m', only the items m with $m.v_0 \leq n \leq m.v_1$ are considered.

In symbolic interpretation expressions are never resolved to concrete or abstracted variable values, instead, a resolution stops if the expression only contains literals (including base addresses which are considered as string literals), operators and symbols from a given set V and with a specific version range $n_0 \leq n \leq n_1$. A typical resolution variant is to take V as the set of base addresses, function call parameters and global input variables, and specify $n = 0$, meaning *"resolve expression until it only contains literals, operators and input variables in their initial version"*. The constraints of the memory items involved are part of the resolution result ρ, so in general ρ is of the form

$$\rho = \textbf{if } c_{11} \wedge \ldots \wedge c_{1k_1} \textbf{ then } e_1 \textbf{ elseif } c_{21} \wedge \ldots \wedge c_{2k_2} \textbf{ then } e_1 \ \ldots \textbf{else } e_\ell$$

with expressions $e_i \in Expr(V)$, that is, without version information. Examples for handling memory items in S_S are given in Section 6.

Symbolic interpretation is performed according to rules of the pattern

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2}{(n_1, n, mem) \longrightarrow_G (n_2, n+1, mem')},$$

so a transition can be performed on symbolic level whenever a corresponding edge exists in the control flow graph [5]. To illustrate the effect of symbolic transitions on the state space S_S we present three transition rules explaining stack variable definition, assignment to a variable (selector) and assignment to a de-referenced pointer.

[5] It may turn out, however, on abstract or concrete interpretation level, that such a transition is *infeasible* in the sense that no valuation of inputs exists where the constraints of all memory items involved evaluate to `true`.

function $up_{=}(sel : Selectors;\ expr : Expr;\ n : \mathbb{N}_0;\ h : M\text{-}Item^*;\ g : Expr) : M\text{-}Item^*$
  $m' := (n+1, \infty, \&\beta(sel), \tau(stack, sel), \omega_A(sel), 8 \cdot \texttt{sizeof(}sel\texttt{)}, (expr, n), (g, n));$
  $up_{=} := up(m', n, h);$
end

Fig. 2. Effect of normal assignments on history of memory items.

(1) A *stack variable definition* , $n_1 =_{\text{def}}$ `typex x;` only affects the current stack frame. Value expression $\top$ marks that the value is still undefined.

$$mem' = (mem.data, mem.heap, front(mem.stack) \frown \langle last(mem.stack) \frown \langle m \rangle\rangle$$
$$m = (n+1, \infty, \&x, \texttt{typex}, 0, 8 \cdot \texttt{sizeof(typex)}, \top, (g, n))$$

(2) The effect of an *assignment to a stack variable*, $n_1 =_{\text{def}}$ `sel = expr;` affects the current stack frame only:

$$mem' = (mem.data, mem.heap, front(mem.stack) \frown \langle h' \rangle)$$
$$h' = up_{=}(sel, expr, n, last(mem.stack), g)$$

Function $up_{=}()$ (Fig. 2) specifies (1) how a new memory item m' is created for the stack frame history, carrying the right-hand side expression as its value and the CFG guard condition as validity constraint and (2) which memory items m have to be *invalidated* due to the new assignment, possibly leading to the creation of "replacements" for these m involving new constraints. The details of this invalidation/creation process are specified in function $up()$ (Fig. 3): All memory items m matching with the new item m' with respect to base address and validity information have to be invalidated. It may be the case, however, that m' "overwrites" only a portion of m. As a consequence, it has to be specified that the "remains" of m not affected by the assignment m' are still valid. Therefore a new memory item m_1 is created and its constraint specifies that outside the range of m', the old m valuation still exists. Observe that the constraint of m_1 always evaluates to `false` if m and m' are of the same type and have the same offset. This indicates that m_1 is infeasible, so m is completely overwritten.

The effect of assignments to variables in the data segments are specified analogously; they affect the *mem.data*-portion of the memory state.

(3) An *assignment to a de-referenced pointer*, $n_1 =_{\text{def}}$ `*p = expr;` may affect the data segment, heap or stack, depending on the potential target addresses `p` points to. The details are specified by function $up_{=_p}$ (Fig. 4).

$$mem' = up_{=_p}(p, expr, n, mem, g)$$

At first, a list ml of all possible pointer targets is generated, using auxiliary function $\gamma()$ (Fig.5): Depending on the valuation of different constraints associated with different memory items, p may point to one or more locations in stack, data segment or heap. For each of these possible situations, ml contains the new memory item for the respective pointer target. The effect of each new item on the invalidation of existing items and creation of new ones is performed again as specified by $up()$ and explained above.

5 Abstract Interpretation of GIMPLE-Programs

Based on the symbolic interpreter introduced in the preceding section it is now possible to construct *a variety of abstract interpreters* according to the following

```
function up(m' : M-Item; n : ℕ0; h : M-Item*) : M-Item*
  u := ⟨ ⟩; w := ⟨ ⟩;
  for m = last(h) downto head(h) do
    if (m.v1 = ∞ ∧ m'.a = m.a) then
      m1 := (n + 1, ∞, m.a, m.t, o, l, m.val,
             m.c ∧ m'.c ∧ 0 < l ∧ m.o ≤ o ∧ o < l ∧ l ≤ m.l ∧ (o + l ≤ m'.o ∨ m'.o + m'.l ≤ o));
      m.v1 := n;
      u := ⟨m1⟩ ⌢ u;
    endif
    w := ⟨m⟩ ⌢ w;
  enddo
  up := w ⌢ u ⌢ ⟨m'⟩;
end
```

Fig. 3. Effect of new memory item on history $h \in M^*$.

```
function up_{=p}(p : Symbols; expr : Expr; n : ℕ0; mem : M; g : Expr) : M
  h_d := mem.data; h_h := mem.heap; h_s := last(mem.stack);
  ml := γ(p, expr, n, mem, g);
  forall m' in ml do
    if (σ(m') = data) then h_d := up(m', n, h_d)
    elseif (σ(m') = heap) then h_h := up(m', n, h_h)
    else h_s := up(m', n, h_s);
  enddo
  up_{p=} := (h_d, h_h, front(mem.stack) ⌢ ⟨h_s⟩);
end
```

Fig. 4. Effect of assignments to de-referenced pointers on history of memory items.

```
function γ(p : Symbols; expr : Expr; n : ℕ0; mem : M; g : Expr) : M-Item*
  ml := ⟨ ⟩;
  if σ(p) = data then h := mem.data else h := last(mem.stack);
  forall m_p in h do
    if m_p.a = &p ∧ m_p.v0 ≤ n ≤ m_p.v1 then
      pl := ξ(m_p.val, mem);
      forall q in pl do
        a := base address from expression pl;
        o := offset expression from expression pl;
        c := conjunction over all conditions of memory items occurring in pl;
        m' := (n + 1, ∞, a, τ⃗(p), o, 8 · sizeof(τ⃗(p)), (expr, n), c ∧ (g, n));
        ml := ml ⌢ ⟨m'⟩;
      enddo
    endif
  enddo
  γ := ml;
end
```

Fig. 5. Function γ finds list of memory items potentially affected by assignment to de-referenced pointer.

rules: (1) For every datatype t in the concrete program component chose a suitable abstraction lattice $(L(t), \sqsubseteq)$, so that a Galois connection $(\mathbf{P}(t), \subseteq) \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} (L(t), \sqsubseteq)$ exists. (2) Lift each operation $\diamond$ defined on t to $L(t)$ by means of the canonic construction $\diamond_L : L(t) \times L(t) \to L(t)$; $p_1 \diamond_L p_2 =_{\text{def}} (p_1{}^{\triangleleft} \diamond_{\mathbf{P}} p_2{}^{\triangleleft})^{\triangleright}$. In this definition, $\diamond_{\mathbf{P}}$ denotes the canonic lifting of $\diamond$ to the powerset lattice over t:

```
function ξ((expr, n) : Expr × ℕ0; mem : M) : M-Item − Expr*
  el := ⟨expr⟩;
  e := head(el);
  while e is not resolved to base address plus M-Item-expression for offset do
    x := next unresolved identifier from e;
    h := if σ(x) = stack then last(mem.stack) else mem.data;
    for m := last(h) downto head(h) do
      if m.a = &β(x) ∧ m.v0 ≤ n ≤ m.v1 then
        e1 := e;
        In e1: exchange each occurrence of x by m;
        el := el ⌢ ⟨e1⟩;
      endif
    enddo
    el := tail(el);
    e := head(el);
  enddo
  ξ := el;
end
```

Fig. 6. Function ξ finds list of base addresses potentially associated with a pointer.

$a_1 \Diamond_{\mathbf{P}} a_2 =_{\text{def}} \{x_1 \Diamond x_2 \mid x_i \in a_i, i = 1,2\}$ (3) Having defined all abstraction lattices $L(t)$, lift all Boolean operators
$\triangle : t \times t' \to \mathbb{B}$ to $[\triangle] : L(t) \times L(t') \to L(\mathbb{B})$ by

$$p_1[\triangle]p_2 = \begin{cases} \top \text{ if } \{x_1 \triangle x_2 \mid x_1 \in {p_1}^{\triangleleft}, x_2 \in {p_2}^{\triangleleft}\} = \{\texttt{false}, \texttt{true}\} \\ \texttt{false} \text{ if } \{x_1 \triangle x_2 \mid x_1 \in {p_1}^{\triangleleft}, x_2 \in {p_2}^{\triangleleft}\} = \{\texttt{false}\} \\ \texttt{true} \text{ if } \{x_1 \triangle x_2 \mid x_1 \in {p_1}^{\triangleleft}, x_2 \in {p_2}^{\triangleleft}\} = \{\texttt{true}\} \end{cases}$$

(4) Lift the symbolic state space $S_S = N(P) \times \mathbb{N}_0 \times M$ defined above to its lattice representation $S_L = N(P) \times \mathbb{N}_0 \times L(M)$, where $L(M)$ is the interpretation of memory items over the respective abstraction lattices chosen for offsets, length, values and constraints. (5) The transition rules for the abstract interpretation semantics over S_L are of the form

$$\frac{n_1 \xrightarrow{g}_{CFG} n_2, (n_1, n, mem) \longrightarrow_G (n_2, n+1, mem'), L(mem) \models_L (g \neq \texttt{false})}{(n_1, n, L(mem)) \longrightarrow_L (n_2, n+1, L(mem'))}$$

where $L(mem)$ denotes the lattice interpretation of memory items. Informally speaking, an abstract transition between CFG nodes n_1 and n_2 with changes in abstract memory valuations from $L(mem)$ to $L(mem')$ is possible in S_L if (a) there exists a corresponding edge in the CFG, (b) the lattice valuation of the guard condition g is `true` or $\top$ and (c) the collection of memory items changes from mem to mem' in the symbolic interpretation.

6 Application Example

The following example illustrates some of the advantages obtained by the higher flexibility resulting from the interplay between symbolic and abstract interpretation. Consider the GIMPLE function[6] shown in Fig. 7 and an associated invocation $x = f(i_0, z_0)$; Applying the symbolic interpretation rules described in Section 4 for the two possible paths through the function results in the symbolic state of the stack

[6] Observe that in contrast to C/C++, GIMPLE always uses byte values in pointer arithmetic. As a consequence, we find assignment `q = p + 4*i;` in line 4, whereas we would write `q = p + i;` in the corresponding C/C++ program.

frame as shown in the list of memory items on the right-hand side of Fig. 7, valid at function return in line 11. Consider the following verification goals: *(Goal 1): $f()$ only assigns to valid de-referenced pointers.*, *(Goal 2): $f()$ never returns an undefined value.*

```
0   float f(int i, float z) {
1     float *p, *q;
2     float a[10];
3     p = &a;
4     q = p + 4*i;
5     if ( 0 < z ) {
6       *q = 10 * z;
7     else {
8       *q = 0;
9     }
10    return a[i];
11  }
```

Line No. Resulting M-Item

0. $(1, \infty, \&i, int, 0, 32, (i_0, 0), \texttt{true})$
0. $(1, \infty, \&z, float, 0, 32, (z_0, 0), \texttt{true})$
0. $(1, 6, \&xReturn, float, 0, 32, \bot, \texttt{true})$
1. $(2, 3, \&p, float*, 0, 32, \bot, \texttt{true})$
1. $(2, 4, \&q, float*, 0, 32, \bot, \texttt{true})$
2. $(3, 5, \&a, float, 0, 320, \bot, \texttt{true})$
3. $(4, \infty, \&p, float*, 0, 32, \&a, \texttt{true})$
4. $(5, \infty, \&q, float*, 0, 32, (p + 4 \cdot i, 4), (0 \leq i < 10, 4))$
6. $(6, \infty, \&a, float, (32 \cdot i, 5), 32, (10 \cdot z, 5), (0 < z, 5))$
6. $(6, \infty, \&a, float, \underline{o}, \underline{l}, \bot, (0 < z \wedge 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i \vee 32 \cdot i + 32 \leq \underline{o}), 5))$
8. $(6, \infty, \&a, float, (32 \cdot i, 5), 32, 0, (z \leq 0, 5))$
8. $(6, \infty, \&a, float, \underline{o}, \underline{l}, \bot, (z \leq 0 \wedge 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i \vee 32 \cdot i + 32 \leq \underline{o}), 5))$
10. $(7, \infty, \&xReturn, float, 0, 32, (a[i], 6), \texttt{true})$

Fig. 7. GIMPLE Code sample and associated symbolic interpretation result.

Alternative 1: Interpretation with is-defined and interval lattices.

Chose lattice $L_D = (\{\bot, \Delta, \top\}, \sqsubseteq)$ with $\bot \sqsubseteq \Delta \sqsubseteq \top$ as an appropriate abstraction for checking well-definedness of `float z; float a[10];` (Δ stands for *is-defined*, $\bot$ for *is-undefined*). For checking pointer addresses we abstract integers to intervals over $\mathbb{Z}$: $L_I = (\mathbb{I}(\mathbb{Z}), \subseteq)$. With these lattices, we now perform the corresponding abstract interpretation on the history of memory items in Fig. 7, each time resolving the associated to symbols down to constants, base addresses or input variables i_0, z_0 as explained in Section 4. Additionally we assume that a precondition $i_0 \in [3, 5]$ has been asserted. Then the abstract interpretation results in

0. $(1, \infty, \&i, L_I, 0, 32, [3, 5], \texttt{true})$
0. $(1, \infty, \&z, L_D, 0, 32, \Delta, \texttt{true})$ (z is well-defined, since it is initialised with input z_0)
0. $(1, 6, \&xReturn, L_D, 0, 32, \bot, \texttt{true})$
1. $(2, 3, \&p, L_I, 0, 32, [-\infty, +\infty], \texttt{true})$
1. $(2, 4, \&q, L_I, 0, 32, [-\infty, +\infty], \texttt{true})$
2. $(3, 5, \&a, L_D, 0, 320, \bot, \texttt{true})$
3. $(4, \infty, \&p, L_I, 0, 32, [\&a, \&a], \texttt{true})$ (symbolic single-point interval $[\&a, \&a]$)
4. $(5, \infty, \&q, L_I, 0, 32, \&a + 4 \cdot [3, 5], \texttt{true})$ ($([0, 0][\leq][3, 5][<][10, 10])$ is true in L_I
6. $(6, \infty, \&a, L_D, 32 \cdot [3, 5], 32, \Delta, \top)$ ($0 < \Delta$ evaluates to $\top$, $10 \cdot \Delta$ evaluates to Δ over L_D)
6. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot [3, 5] \vee 32 \cdot [3, 5] + 32 \leq \underline{o}))$
8. $(6, \infty, \&a, L_D, 32 \cdot [3, 5], 32, \Delta, \bot)$ ($\{0\}^{\triangleright} = \Delta$)
8. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot [3, 5] \vee 32 \cdot [3, 5] + 32 \leq \underline{o}))$
10. $(7, \infty, \&xReturn, L_D, 0, 32, (a[[3, 5]], 6), \texttt{true})$ (not yet resolved – see next paragraph)

Now we apply the resolution rules to 10: First it is noted that $a([[3, 5]], 6)$ matches all memory items of the form

$$m = (v_0, v_1, \&a, L_D, 32 \cdot [3, 5], 32, val, c),\ v_0 \leq 6 \wedge 6 \leq v_1$$

As a consequence the valuation candidates are those from lines 6. and 8. above. We only have to investigate the feasibility of memory items with undefined valuation $\bot$, so it remains to show that

$$0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot [3,5] \vee 32 \cdot [3,5] + 32 \leq \underline{o}) \wedge \underline{o} = 32 \cdot [3,5] \wedge \underline{l} = 32$$

has no solution; this proof obligation is simplified to showing that no solution of

$$[3,5] + 1 \leq [3,5] \vee [3,5] + 1 \leq [3,5]$$

exists. Unfortunately this predicate evaluates to $\top$ in L_I because we can select (different) numbers from $[3,5]$ in each of its occurrences so that the predicate evaluates either to `true` or to `false`. As a consequence it is necessary to perform 2 partitioning steps of the i_0 interval valuation $[3,5]$ into $[3,3]$ $[4,4]$, $[5,5]$, in order to prove that this predicate is always `false`.

Alternative 2: Interpretation with is-defined and predicate lattice.

As we have seen in the discussion of alternative 1 above, the interval lattice is suitable for proving well-definedness of pointer de-referencings but is quite inefficient to prove the crucial step for well-definedness of the return value. We can fix this by taking the solution of verification goal 1 as constructed above, but using another lattice to represent pointer and integer expressions for discharging goal 2: Let L_P the lattice of predicates over programming variables, together with their comparison operators [7]. Use L_D as above. Abstract interpretation now results in

0. $(1, \infty, \&i, L_P, 0, 32, i = i_0, \texttt{true})$
0. $(1, \infty, \&z, L_P, 0, 32, \Delta, \texttt{true})$
0. $(1, 6, \&xReturn, L_D, 0, 32, \bot, \texttt{true})$
1. $(2, 3, \&p, L_P, 0, 32, \bot, \texttt{true})$
1. $(2, 4, \&q, L_P, 0, 32, \bot, \texttt{true})$
2. $(3, 5, \&a, L_D, 0, 320, \bot, \texttt{true})$
3. $(4, \infty, \&p, L_P, 0, 32, p = \&a, \texttt{true})$
4. $(5, \infty, \&q, L_P, 0, 32, q = \&a + 4 \cdot i_0, \texttt{true})$
6. $(6, \infty, \&a, L_D, 32 \cdot i_0, 32, \Delta, \top)$
6. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i_0 \vee 32 \cdot i_0 + 32 \leq \underline{o}))$
8. $(6, \infty, \&a, L_D, 32 \cdot i_0, 32, \Delta, \top)$
8. $(6, \infty, \&a, L_D, \underline{o}, \underline{l}, \bot, 0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i_0 \vee 32 \cdot i_0 + 32 \leq \underline{o}))$
10. $(7, \infty, \&xReturn, L_D, 0, 32, (a[i_0], 6), \texttt{true})$ (not yet resolved – see next paragraph)

Now, for the resolution of $(a[i_0], 6)$, all memory items of the form

$$m = (v_0, v_1, \&a, L_D, 32 \cdot i_0, 32, val, c),\ v_0 \leq 6 \wedge 6 \leq v_1$$

match, and the condition for returning an undefined value is

$$0 < \underline{l} \wedge 0 \leq \underline{o} \wedge \underline{o} + \underline{l} \leq 320 \wedge (\underline{o} + \underline{l} \leq 32 \cdot i_0 \vee 32 \cdot i_0 + 32 \leq \underline{o}) \wedge \underline{o} = 32 \cdot i_0 \wedge \underline{l} = 32$$

which – applying the rules on term replacement and arithmetics in L_P – boils down to $i_0 + 1 \leq i_0$ which is obviously `false`.

7 Conclusion

We have described techniques for concrete and abstract interpretation of C/C++ programs represented in GIMPLE, which basically produces a control flow graph model for each C/C++ function or method. The results are implemented in a tool and they are currently applied for integrated module testing and static analysis of safety-critical embedded systems software in the railway and avionic domains. Applications in the field of automotive control are currently prepared; they focus,

[7] More formally, the *quantifier-free Presburger formulae* over program variables are suitable for our purpose because efficient solvers exist for problems of this type [13].

however, on model-based test case generation. Due to the intermediate model representation of the tool which uses the same class of hierarchic transition systems for code (control flow graph) and model (e. g. UML 2.0 Statechart) representation, the test case generation mechanisms are the same for code-based and model-based testing. Currently a correctness proof for the abstract interpretation semantics constructed according to the rules given in Section 5 is elaborated: We show that application of these rules always result in a valid abstract interpretation semantics according to Definition 2.1.

References

[1] Bahareh Badban, Martin Fränzle, Jan Peleska, and Tino Teige. Test automation for hybrid systems. In *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*, Portland Oregon, USA, November 2006.

[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 1–24, Tokyo, Japan, LNCS, December 6–8 2006. Springer, Berlin. (to appear).

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[4] Bruno Blanchet et. al. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. AE. Mogensen et al., editor, *The Essence of Computation*, volume 2566, pages 85–108, 2002.

[5] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna - a static model checker. In *Proceedings of 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Bonn, Germany, 2006.

[6] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex bo olean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 2007.

[7] GCC, the GNU Compiler Collection. The GIMPLE family of intermediate representations. See http://gcc.gnu.org/wiki/GIMPLE.

[8] Helge Löding. Behandlung komplexer Datentypen in der automatischen Testdatengenerierung. Master's thesis, University of Bremen, May 2007.

[9] Jan Peleska and Helge Löding. *Static Analysis By Abstract Interpretation*. University of Bremen, Centre of Information Technology, 2008. available under http://www.informatik.uni-bremen.de/agbs/lehre/ws0708/ai/saai_script.pdf.

[10] Jan Peleska, Helge Löding, and Tatiana Kotas. Test automation meets static analysis. In Rainer Koschke, Karl-Heinz Rödiger Otthein Herzog, and Marc Ronthaler, editors, *Proceedings of the INFORMATIK 2007, Band 2, 24. - 27. September, Bremen (Germany)*, pages 280–286.

[11] S. Ranise and C. Tinelli. Satisfiability modulo theories. *TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems*, 21(6):71–81, 2006.

[12] Bastian Schlich, Falk Salewski, and Stefan Kowalewski. Applying model checking to an automotive microcontroller application. In *Proc. IEEE 2nd Int'l Symp. Industrial Embedded Systems (SIES 2007)*. IEEE, 2007. ISBN 1-4244-0840-7.

[13] Ofer Strichman. On solving presburger and linear arithmetic with sat. In M.D. Aagaard and J.W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD),*, number 2517 in LNCS, pages 160–170. Springer, 2002.

[14] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. In *Proceedings of the PLDI'04, June 9-11, 2004, Washington, DC, USA*. ACM 1581138075/04/0006.

[15] Verified Systems International GmbH, Bremen. *RT-Tester 6.2 – User Manual*, 2007.

Proving Correctness of an Efficient Abstraction for Interrupt Handling

Herberich, Gerlind[1] Schlich, Bastian[1,2] Weise, Carsten[1,3]

Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany

Noll, Thomas[1]

Software Modeling and Verification Group
RWTH Aachen University
Aachen, Germany

Abstract

This paper presents an approach to the efficient abstraction of interrupt handling in microcontroller systems. Such systems usually operate in uncertain environments, giving rise to a high degree of nondeterminism in the corresponding formal models, which in turn aggravates the state explosion problem. Careful handling of nondeterminism is therefore crucial for obtaining efficient model checking tools. Here, we support this goal by developing a formal computation model and an abstraction method, called interrupt nondeterminism, which instantiates nondeterministic values only if and when this is required by the application code. It is shown how this symbolic technique can be integrated into our explicit CTL model checking tool [mc]square by introducing *lazy states*. A lazy state consists of explicit and symbolic parts and therefore, represents several concrete states. With regard to interrupt handling, we also give a bisimulation relation between the concrete and the abstract state space, thus establishing the correctness of our technique. Furthermore, a case study is presented in which three different programs are used to demonstrate the effectiveness of our method.

Keywords: Model checking, state space abstraction, bisimulation, interrupt handling

1 Introduction

Model checking is recognized by industry as a promising future tool for the analysis of embedded software (e.g., software for microcontrollers). Early model checkers as SMV [4], Spin [8] and Uppaal [9] work on models described in their proprietary specification languages. Re-modeling existing systems in these proprietary formalisms is a huge effort. For existing systems, model checking of higher level programming

[1] Email: "lastname"@cs.rwth-aachen.de

[2] partly funded by a DAAD-Doktorandenstipendium

[3] funded by the UMIC cluster of excellence: http://www.umic.rwth-aachen.de/

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

languages (e.g., C, C++ or Java) is the more efficient approach, but when it comes to embedded software, many problems arise when model checking C programs.

Microcontroller programs written in C usually contain direct hardware accesses or embedded assembly statements. These constructs are not handled by existing C code model checkers (see [16]). Moreover, C code is first compiled into assembly code before it is deployed to the hardware. Hence, the C code is only an intermediate representation. The compiler could introduce errors that cannot be found in the original source code. In assembly code all errors introduced during the complete development process are present. Moreover, in contrast to C code, assembly code has a clean, formal and well documented semantics. Hence, model checking of assembly code (machine code) gets into focus of research, see [1,12,13,17,20].

However, when model checking assembly code state spaces tend to be bigger and the analysis is no longer hardware independent. In order to tackle these problems, we have developed [mc]square [4], which is a discrete, (mostly) explicit state, on-the-fly, Computation Tree Logic (CTL) [4] model checker. It is capable of model checking assembly code written for certain microcontrollers (ATMEL ATmega and Infineon XC167). We did not restrict the set of supported constructs. Hence, [mc]square can handle arbitrary assembly programs for those microcontrollers, supporting both low-level features such as direct or indirect memory access and source-level constructs such as recursion or functions. To address the disadvantage of being hardware-dependent, we developed an extensible architecture, which is described in [18]. To deal with the state explosion problem, we implemented different abstraction techniques in [mc]square. In this paper, we will show how prioritized interrupt levels can be used to abstract away from the concrete state of the interrupt bits of a microcontroller.

The basic idea of the abstraction technique is that in certain states of the processor, certain bits are irrelevant for the execution. These bits can be safely abstracted away by setting them to a *don't care* value, or better said to a *nondeterministic* value, which we will denote by $*$. This introduces so-called *lazy states*. A lazy state is a state that contains explicit and symbolic parts. As a consequence, a lazy state no longer represents a single state, but a set of states. Therefore, [mc]square combines explicit and symbolic techniques. This idea is used for several abstraction techniques within [mc]square, e.g., also for *delayed nondeterminism* (see [15]). In this paper, we focus on abstracting away from the specific values of the interrupt bits.

This paper is structured as follows: We start with the presentation of related work. Then, a basic introduction to [mc]square is given. Section 4 details nondeterminism for interrupts. The subsequent section presents our formal approach to modeling microcontrollers. As an example, the model of the ATMEL ATmega16 microcontroller is detailed. It is shown that nondeterminism of interrupts induces a simulation relation between the concrete trace of the system and the abstracted, nondeterministic traces, thus yielding an over-approximation of the real system behavior. However, while over-approximating the concrete behavior, it is true that for every trace in the abstract space, there is a concrete system which will exhibit

[4] http://www-i11.informatik.rwth-aachen.de/mc_square.html

this behavior. As interrupts depend on the behavior of an external environment which is not under the control of the processor, this means that all errors found in the over-approximation can in fact also be traced back to the real implementation. After that, a case study is summarized which demonstrates the effect of interrupt nondeterminism on the state space size. In the end a conclusion is drawn and some potential directions for future improvements are shown.

2 Related Work

Motivated by the observation that usually memory is the limiting factor in the application of model checking, many approaches have been developed to combat the state explosion problem (see [4] for an overview). The abstraction technique presented in this paper, *Interrupt NonDeterminism*, is dynamically applied at runtime. To the best of our knowledge, no comparable approach has been developed so far to control the effect of interrupts in modeling embedded systems.

There is, however, a verification method for concurrent systems called *narrowing* which is based on a similar idea, and which is described in [14]. Here, the states and transitions of the system are symbolically represented by terms and rewriting steps, respectively. Terms can contain variables to abstract from details of the system state which currently are not "interesting", but which can later be expanded by substitution steps if necessary. Thus, in some sense, variables correspond to the nondeterministic values in our approach.

Another direction of work which is worth mentioning is the concept of *lazy evaluation* in functional programming languages [10], which computes a function argument only if it is accessed in the function body. Moreover the paper [3] studies the implementation of nondeterministic choice in this setting and refers to the problem of copying nondeterministic values, which is also the reason for over-approximation in our model.

Symbolic simulation is another technique that is similar to the technique applied in [mc]square. Here, symbolic values are used in place of explicit values. In our approach parts of the states used can be symbolic, but whenever the simulator or the model checker needs to access symbolic parts of a state, these parts are instantiated, and hence, become explicit. All parts of a state that are not accessed remain symbolic. In [2], a symbolic simulator is used to verify hardware systems. Whenever an X (denoted by * in our approach) is accessed and a value is needed, new symbolic variables are added and simulation has to be repeated. In our approach a dynamic refinement is conducted. There are some approaches combining *explicit* and *symbolic executions* (cf. [6,22]), but these approaches employ explicit execution and symbolic execution in parallel.

Other model checkers that handle machine languages or languages that are similar to machine languages are *Java PathFinder* (*JPF*) [24], *StEAM* [11], and *Estes* [13], all being explicit model checkers as is [mc]square. JPF accepts Java bytecode and employs collapsing techniques for efficiently storing states. Our experiments have shown that such methods do not pay off in the case of [mc]square since its states have a less complex structure. Another difference is that JPF has to deal with concurrent processes and therefore employs abstraction techniques such as

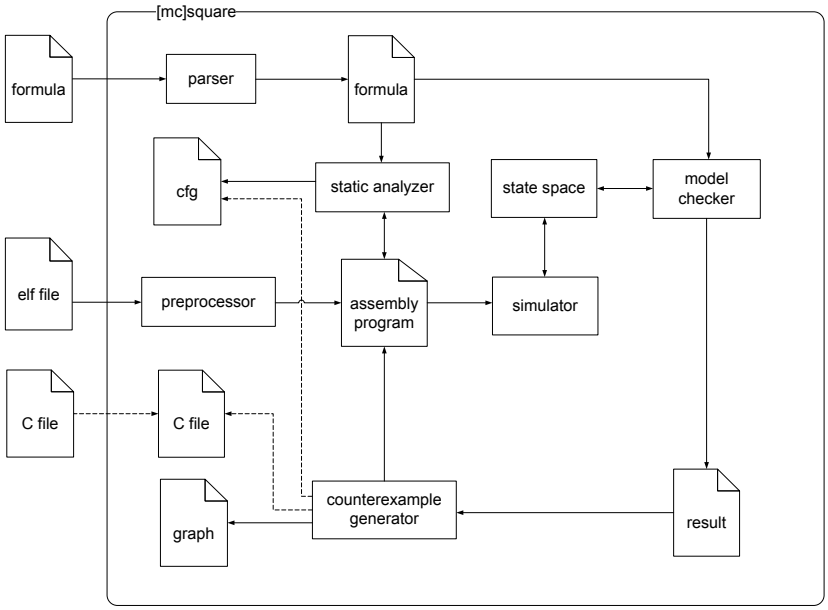

Figure 1. Process used in [mc]square

partial order reduction, which cannot be done in [mc]square. Moreover the memory model used within JPF makes it possible to apply symmetry reduction techniques. Again, this is not possible in [mc]square because the order of data within memory is important. StEAM model checks bytecode for the Internet C Virtual Machine. In this approach an existing Virtual Machine (VM) is monitored and model checking is conducted on the states created by this VM. Estes model checks assembly code for a certain processor. Similar to StEAM, it uses an existing VM (the GNU debugger) to create the state space. In our approach, we concentrate on the creation of the state space, that is, we concentrate on the domain-specific abstractions implemented within the simulator. We do not want to use existing simulators as we think that significant savings can be achieved by a tailored implementation. In contrast to Estes we abstract from time because the state explosion observed when temporal aspects are taken into account, i.e., real-time model checking (cf. [9]) is performed, is too big.

3 Introduction to [mc]square

This section gives an introduction to [mc]square, which is a discrete, (mostly) explicit state, on-the-fly, CTL model checker. "Mostly explicit" means that [mc]square uses explicit model checking algorithms, but combines them with symbolic techniques. We call the new states that combine explicit and symbolic parts *lazy states*. Whenever a symbolic part of a lazy state is accessed, the nondeterminism is resolved automatically. [mc]square works on assembly code written for certain microcontrollers (ATMEL ATmega and Infineon XC167). More information about [mc]square can be found in [17,18].

The process that is applied in [mc]square is depicted by Fig. 1. First, the user inputs the program as an Executable and Linking Format (ELF) file and the

specification as a CTL formula. If the C code is available, the user can also provide the corresponding file. The formula is parsed and transformed into a formula object, which is utilized by the static analyzer and the model checker component. The ELF file is preprocessed and converted into an human readable assembly program.

Then, the static analyzer component starts inspecting the assembly program. During this analysis, it uses information from the formula object (registers, variables and memory locations used within the atomic propositions) to preserve validity of the results. In the first step of the static analysis, a Control Flow Graph (CFG) of the assembly program is created. This CFG is inter alia used by the counterexample generator to present counterexamples or witnesses. In the end, the static analyzer adds annotations to the assembly program which are used by the simulator to reduce the size of the state space.

After that, model checking starts. First, the model checker requests the initial state from the state space generator. It checks this state for certain parts of the formula, and depending on the result of this check, it requests successor states of this state. Then, it again checks these states for specific parts of the formula. This process continues until a goal state is reached (proving or disproving the validity of the formula) or the complete state space is built. The model checking algorithm used is taken from [7]. A first version of this algorithm was presented in [23].

Whenever successors of a state are requested that are not created yet, the state space generator uses the simulator to on-the-fly create the needed states. To do so, it passes the current state to the simulator and calls a `step()` method. The simulator creates all possible successors of this state including, e.g., occurrences of interrupts, different input values from the environment etc. If, e.g., an instruction `IN R18 PINA` reads input from the environment into register `R18`, then all possible values might occur, which results in 256 successors states (all values between 0 and 255). Also, if a specific interrupt is enabled in a state of the system, then the interrupt might potentially occur, and thus both successor states – one taking the interrupt, one with the usual program execution step – must be considered by [mc]square.

As the last step of analysis, the counterexample generator derives a counterexample or a witness depending on the formula checked and the result of the model checking process. This counterexample/witness is then presented in the assembly code, in the C code, as a state space graph, or in the CFG of the assembly code. Hence, the user can choose the representation that suits his requirements best to find the error. As some abstraction techniques used in [mc]square only preserve ACTL [5], the user can use this representation to check whether a witness of an non-ACTL formula is a feasible one.

During state space building [mc]square uses different abstraction techniques to minimize the size of the state space. It is important to notice that all these abstractions lead to a safe over-approximation of the concrete state space, establishing a simulation relation between the concrete and the abstract states (and thus preserving the validity of ACTL formulae). One of these abstraction techniques is interrupt nondeterminism, which is detailed in this paper.

[5] the universal fragment of CTL; see [5]

	i0	i1	i2	i3
source active	+	+	+	+
interrupt enabled	+	+	-	+
interrupt flag	*	*	*	*

Table 1
Four interrupts exemplifying nondeterministic interrupts.

i0	i1	i2	i3	
0	0	0	0	→ no interrupt
0	0	0	1	→ interrupt 3
0	0	1	0	→ no interrupt
0	0	1	1	→ interrupt 3
0	1	0	0	→ interrupt 1
...	...	...	...	...
1	1	1	0	→ interrupt 0
1	1	1	1	→ interrupt 0

i0	i1	i2	i3	
1	*	*	*	→ interrupt 0
0	1	*	*	→ interrupt 1
0	0	*	1	→ interrupt 3
0	0	*	0	→ no interrupt

Table 2
Deterministic and nondeterministic evaluation of interrupts.

4 Interrupt Nondeterminism

Interrupt nondeterminism abstracts away from interrupts which are below the current interrupt level to reduce the state space for interrupt handling. Whenever an interrupt at level ℓ is taken, all interrupt flags for levels below ℓ can be set to the nondeterministic value $*$. When not using the abstraction, there are 2^s different possible combinations where s is the number of active interrupt sources. Abstraction reduces this number to $a - 1$ where $a \leq s$ is the number of enabled interrupts, which yields a drastical saving in terms of successor states.

However, even when interrupts are disabled or masked, it still is possible to check their values, or enable them again. In this cases, the analysis again must consider the actual values of the interrupt bits. Thus, we need to consider that interrupt bits are tested or set, and then make their value explicit again. This step is called the *instantiation* of a bit, and can occur in two ways: if the bit is tested, the simulation needs to take both cases into account: that its value is 0 or 1. Two successor states must be created, one for each possibility – or even more, if several bits are tested in parallel. If the bit is set to a specific value, then the bit is just set to this value in the successor state.

In Tab. 1 an example configuration of four interrupts is shown. The sources of all four interrupts are active, but only three of these interrupts are enabled. In a concrete simulation, all 16 value combinations shown in the left part of Tab. 2 are written to the interrupts flag registers. In contrast, only 4 combinations are written to the interrupt flag registers when using interrupt nondeterminism. These four combinations are shown in the right part of Tab. 2.

Note that if an interrupt bit occurs in the formula tested by the model checker, then that bit must be excluded from interrupt nondeterminism. This is a simple implementation issue not detailed in this paper.

Here, we described handling of interrupts using a static interrupt priority table for the ATMEL ATmega microcontrollers. Nondeterministic interrupts also work using a dynamic interrupt priority table because at the time of interrupt handling,

the priorities are fixed as the interrupt table can only be changed by instructions. Moreover whenever interrupts are handled, no instruction is executed at that moment.

5 The Formal Model

Delayed nondeterminism (DND) has been presented in [15] first, where it is proved that DND preserves a simulation relation. Here, we show that the restriction of this technique to interrupt handling, *interrupt nondeterminism*, even preserves bisimulation. Thus it is not only sound with respect to "path-universal" logics such as ACTL but yields a correct abstraction with respect to the full CTL logic.

5.1 Basics of the Model

Our formal proof for the correctness of the nondeterministic approximation of interrupt handling uses a formal model defining the behavior of our system by a semantic function. The intuition behind the formal model is described in the section.

The model works on the following basic data types:

- the basic data in the model are bits $\mathbb{B} := \{0, 1\}$ and bytes $\mathbb{C} := \mathbb{B}^8$,
- to model nondeterminism, we use the "don't care" bit $*$, and extend the set of bits by this value: $\mathbb{B}_* := \mathbb{B} \cup \{*\}$, and similarly $\mathbb{C}_* := \mathbb{B}_*^8$.

The microcontroller consists of a control and a data space:

- The data space is modeled by an ordered set of addresses A, and an association of (nondeterministic) byte values to these addresses $v : A \to \mathbb{C}_*$. Here we assume w.l.o.g. that each address comprises m bytes, that is, $A = \{0, \ldots, 2^{8m} - 1\}$. The set of all data spaces is denoted by V.
- As we often need to select specific bits from bytes, we use $A_{bit} := A \times \{0, \ldots, 7\}$ as the address space for bits, and then interpret v as $v : A_{bit} \to \mathbb{B}_*$, where $v(a) = v(a, 7) \ldots v(a, 0)$.
- The control (code) space is modeled by a set of locations Q, which is a linearly ordered set, and a mapping of the locations to instructions `Ins`. More details on instructions can be found below.

The above model is a straightforward implementation of a Harvard Computer, where we allow certain bits to be *nondeterministic* in the model, i.e., in a specific snapshot we do not know the status of certain bits.

We will now define a transition relation, which describes the behavior of this machine. The basic idea is that the machine cycles through instructions, and that each instruction defines the read- and write accesses to the memory as well as the control location of the next instruction (which is the following location in the normal case, but a different one in case of jumps).

The machine we describe is used to control some environment. The communication with the environment happens via ports, which are lines sending bits into the environment or receiving bits from the environment. We assume the usual memory-mapped I/O, so the ports are special addresses within the memory of the machine.

The setting of the input ports is described by an environment specification `env`, which is a sequence of guarded assignments to the ports. The guarded assignments are executed each time before an instruction is carried out.

Further, the machine has a set of interrupt routines. The interrupt routines are described by the *interrupt handler* `IH`, which is a sequence of guarded assignments, where each guarded assignment has a continuation, i.e. a location to which the machine will jump when the interrupt routines are triggered. The guarded assignments describe the interrupt logic of the machine: the guard specifies when the interrupt is taken (e.g. when a certain interrupt is enabled and the interrupt flag is set) and the assignments model the interrupt logic, e.g. storing of the current program counter on the system stack happens here. Further, the interrupt routines are prioritized, i.e. they are executed in a specific order from highest to lowest priority, and the routine with the highest priority and an enabled guard is executed first.

The interrupt handler is executed after the environment specification and before the respective instruction. This way, a synchronized, clock-cycle triggered execution of the machine is guaranteed.

We now define the syntax and semantics of assignments, guards and expressions. An assignment takes the form $x \leftarrow e$, where x is an *address expression* and e is a *value expression*. Let Z bet the set of all assignments.

For address expressions, we have $x ::= a \mid a\downarrow + d \mid a[b]$ where $a \in A$, $b \in \{0, \ldots, 7\}$, and $d \in \mathbb{Z}$. The first case is a direct byte access, the second is an indirect byte access with displacement d, the third is accessing the bth bit at address a. Let X be the set of all address expressions.

A value expression is either an address expression, or an operation on value expressions $e ::= op(e_1, \ldots, e_n)$. Possible domains are $\mathbb{B}$ and $\mathbb{C}$ both for argument and result. Let E be the set of all value expressions.

A guard is a test on the equality of two value expressions, boolean combinations thereof or negation: $g ::= e_1 == e_2 \mid g_1 \wedge g_2 \mid g_1 \vee g_2 \mid \neg g_1$, where e_i are value expressions and the g_i are guards. Let G be the set of all guards.

5.2 Semantics in the Deterministic Case

We can now give a semantics for the behavior of the microcontroller for the deterministic case.

The semantics $[\![e]\!] : V \to \mathbb{C} \cup \mathbb{B}$ of a value expression is defined by $[\![a]\!](v) := v(a)$, $[\![a\downarrow + d]\!](v) := v(a' + d)$ where a' denotes the address which is referred to by the m bytes stored at a: $a' := v(a) \cdot 2^{8(m-1)} + \ldots + v(a + m - 1) \in A$, $[\![a[b]]\!](v) := v(a, b)$, and $[\![op(e_1, \ldots, e_n)]\!](v) = op([\![e_1]\!](v), \ldots, [\![e_n]\!](v))$.

Let $v\{a/e\}$ be the valuation v' where $v'(a) = e$ and $v'(a') = v(a')$ else. Then assignments have the semantics $[\![x \leftarrow e]\!] : V \to V$ where $[\![a \leftarrow e]\!](v) = v\{a/[\![e]\!](v)\}$, $[\![a\downarrow + d \leftarrow e]\!](v) = v\{a' + d/[\![e]\!](v)\}$ where again $a' := v(a) \cdot 2^{8(m-1)} + \ldots + v(a + m - 1)$, and $[\![a[b] \leftarrow e]\!](v) = v\{(a, b)/[\![e]\!](v)\}$ (assuming type correctness).

Guards have type $[\![g]\!] : V \to \mathbb{B}$, and are defined as $[\![e_1 == e_2]\!](v) = 0$ if $[\![e_1]\!](v)$ not equal $[\![e_2]\!](v)$, and $[\![e_1 == e_2]\!](v) = 1$ else. The boolean operators work as usual: $[\![g_1 \wedge g_2]\!](v) = [\![g_1]\!](v) \wedge [\![g_2]\!](v)$ etc.

On sets, we use the usual operations $*$ and $+$ meaning all finite sequences resp.

all non-empty finite sequences. Let $GA := G \times Z^*$ be the set of guarded assignments, and $GAC := GA \times Q$ the set of guarded assignments with a continuation.

A guarded assignment is enabled when its guard is 1. The semantics of an enabled guard assignment is the execution of the assignment, otherwise the guard assignment does not change the microcontroller's memory. Formally,

$$[\![g : asn]\!](v) = \begin{cases} v & \text{if } [\![g]\!](v) = 0 \\ [\![asn]\!](v) & \text{if } [\![g]\!](v) = 1 \end{cases}$$

The Environment handler $\texttt{env} \in GA^+$ has the form $g_1 : asn_1;\ \ldots\ ; g_n : asn_n$. Its semantics is the consecutive execution of the enabled assignments:

$$[\![g_1 : asn_1;\ \ldots\ ; g_n : asn_n]\!](v) = [\![g_n : asn_n]\!]([\![g_1 : asn_1;\ \ldots\ ; g_{n-1} : asn_{n-1}]\!](v)),\ n \geq 2$$

A guarded assignment with continuation induces a mapping $[\![.]\!] : Q \times V \to Q \times V$, defined as

$$[\![g : asn, q']\!](q, v) = \begin{cases} (q, v) & \text{if } [\![g]\!](v) = 0 \\ (q', [\![asn]\!](v)) & \text{if } [\![g]\!](v) = 1 \end{cases}$$

The interrupt handler $\texttt{IH} \in GAC^+$ has the form $\texttt{IH} = g_1 : asn_1, q_1\ >\ \ldots\ >\ g_n : asn_n, q_n$. We say that an interrupt is *triggered* if its guard is enabled. The function $trigger : GAC^+ \to \mathbb{N}$ defines which interrupt is taken:

$$trigger(\texttt{IH}) = \begin{cases} 0 & \text{if } \forall i.[\![g_i]\!](v) = 0, \\ j & \text{if } [\![g_j]\!](v) = 1, \forall i < j.[\![g_i]\!](v) = 0 \end{cases}$$

and then $[\![\texttt{IH}]\!](v) = [\![g_j : asn_j, q_j]\!](q, v)$ if $trigger(\texttt{IH}) = j$.

The instruction handler $\texttt{Ins} \in Q \times GAC^+$ has the form $q :: g_1 : asn_1, q_1\ >\ \ldots\ >\ g_n : asn_n, q_n$, and will also execute the first enabled assignment:

$$[\![q :: g_1 : asn_1, q_1\ >\ \ldots\ >\ g_n : asn_n, q_n]\!](v) := \\ [\![g_1 : asn_1, q_1\ >\ \ldots\ >\ g_n : asn_n, q_n]\!](q, v)$$

The microcontroller MC is then the tuple $(A, Q, \texttt{Ins}, \texttt{env}, \texttt{IH}, q_0, v_0)$ with start state $q_0 \in Q$ and initial memory $v_0 \in V$. It defines a transition relation on $Q \times V$ by $(q, v) \to (q', v')$ with $v_1 = [\![\texttt{env}]\!](v)$,

$$(q', v') = \begin{cases} [\![\texttt{IH}]\!](q, v_1) & \text{if } trigger(v_1) > 0 \\ [\![\texttt{Ins}]\!](q, v_1) & \text{if } trigger(v_1) = 0 \end{cases}$$

and initial state (q_0, v_0). We will call this behavior of MC the *concrete* behavior.

5.3 Nondeterministic Interrupts

When modeling all possible behaviors of an environment, we would need to take into account all possible settings of the interrupt bits at any time. However, when an interrupt of level ℓ is taken, the lower prioritized interrupts are not important anymore. Thus we can reduce the number of states in the model checker by replacing all interrupt bits with a lower priority by the nondeterministic bit $*$.

To formalize this, we now introduce the nondeterministic transition relation $\rightarrow_{nd}$. This will be done by lifting the deterministic transition relation to nondeterministic evaluations $v : A \to \mathbb{C}_*$.

We start with introducing nondeterminism into our model, using the function $\mathrm{ndet} : (A_{bit} \to \mathbb{B}_*) \times 2^{A_{bit}} \to (A_{bit} \to \mathbb{B}_*)$, which increases the level of nondeterminism:

$$\mathrm{ndet}(M, v) := w \text{ where } w(a,b) := \begin{cases} v(a,b) & \text{if } (a,b) \notin M \\ * & \text{if } (a,b) \in M \end{cases}$$

Assume we have interrupt levels $1, \ldots, n$. Then we have also a set of interrupt bits $IB_1, \ldots, IB_n \in A_{bit}$. Let $\mathrm{IL}(\ell) := \{IB_1, \ldots, IB_{\ell-1}\}$. Let $\mathtt{IH}$ be some interrupt handler with $[\![\mathtt{IH}]\!](q, v) = (q, v')$ in the deterministic case. Then $[\![\mathtt{IH}]\!]_{nd}(q, v) := (q', \mathrm{ndet}(\mathrm{IL}(\ell), v')$ in the nondeterministic case, if $trigger(\mathtt{IH}) = \ell > 0$, and thus interrupt nondeterminism is introduced into the model at this point.

The next step is to lift the semantics of expressions, assignments and guards to nondeterministic valuations, where all bits which are important for the evaluation are still deterministic. To formalize this notion, we defined the set of bits which are read in an expression, assignment, resp. guard. Let $\mathrm{tested} : A \cup E \cup Z \cup G \to 2^{A_{bit}}$ be defined as

$$\begin{aligned}
\mathrm{tested}(a) &:= \{(a,0), \ldots, (a,7)\} \\
\mathrm{tested}(a\!\downarrow + d) &:= \mathrm{tested}(a) \cup \mathrm{tested}(v(a)) \\
\mathrm{tested}(a[b]) &:= \{(a,b)\} \\
\mathrm{tested}(op(e_1, \ldots, e_n)) &:= \mathrm{tested}(e_1) \cup \ldots \cup \mathrm{tested}(e_n) \\
\mathrm{tested}(a \leftarrow e) &:= \mathrm{tested}(e) \\
\mathrm{tested}(e_1 == e_2) &:= \mathrm{tested}(e_1) \cup \mathrm{tested}(e_2) \\
\mathrm{tested}(e_1 \wedge e_2) &:= \mathrm{tested}(e_1) \cup \mathrm{tested}(e_2) \\
\mathrm{tested}(e_1 \vee e_2) &:= \mathrm{tested}(e_1) \cup \mathrm{tested}(e_2) \\
\mathrm{tested}(\neg e) &:= \mathrm{tested}(e)
\end{aligned}$$

Obviously, only the bits in $\mathrm{tested}(e)$ are needed to evaluate the expression e for some valuation v. So for valuations v which are deterministic in $\mathrm{tested}(e)$, the above definitions can be re-used without changes. Let $\rightarrow_x$ be the transition relation induced by this.

Now for a valuation which is nondeterministic in bits needed for evaluation, we will simply instantiate the necessary bits to all possible values. This is formalized by the function: $\mathrm{det} : 2^{A_{bit}} \times (A_{bit} \to \mathbb{B}_*) \to 2^{(A_{bit} \to \mathbb{B}_*)}$, which is capable of decreasing the nondeterminism of a valuation:

$$\begin{aligned}
\mathrm{det}(M, w) := \{v \in A_{bit} \to \mathbb{B}_* \mid \forall (a,b) \in M.v(a,b) \neq * \text{ and} \\
\forall (a,b) \in A_{bit}.w(a,b) \neq * \implies v(a,b) = w(a,b)\}.
\end{aligned}$$

Further, $\mathrm{nb}(w)$ is the set of nondeterministic bits in w, i.e.

$$\mathrm{nb}(w) := \{(a,b) \mid w(a,b) = *\}.$$

Now a semantic function ϕ with $[\![e]\!](v) = v'$ can be extended to a nondeterministic valuation w by setting $[\![e]\!]_{nd}(w) := \{[\![e]\!](v) \mid v \in \det(M, w)\}, M = \text{tested}(e) \cap \text{nb}(w)$, and similarly $[\![e]\!]_{nd}(q, w) := \{[\![e]\!](q, v) \mid v \in \det(M, w)\}$. Note that our semantics now yields a set of valuations as result.

The nondeterministic transition function can then be defined as $(q, w) \rightarrow_{nd} (q', w')$ iff $(q', w') \in [\![MC]\!]_{nd}(q, w)$.

Note that we can lift the definition of tested to the transition relation by the following rules, yielding a notion of tested for our transition relation.

- For consecutive execution of statements, the tested sets need to be joined,
- for statements with priority, only take into account the statements until the first executed.

So whenever we have a deterministic transition $(q, v) \rightarrow (q', v')$, there is a set $M \subseteq A_{bit}$ of the bits referred to by the transition. For all $N \subseteq A_{bit} \setminus M$, the valuation $\hat{w} := \text{ndet}(N, v)$ has all the necessary information for the deterministic relation, and thus we can also apply the definition, yielding a transition $(q, \hat{w}) \rightarrow_x (q', \hat{v})$. As this transition results from reading and writing the same bits as for $(q, v) \rightarrow (q', v')$, there must be $N' \subseteq N$ and $\hat{v} = \text{ndet}(N', v')$. It must not be $N = N'$, as nondeterministic bits can be assigned a value by the transition.

5.4 Modeling the ATMEL ATmega16

Now we will show how to use the general framework to model the ATMEL ATmega16 microcontroller. The ATMEL ATmega16 has a 16K flash memory for program code, which corresponds to the location set Q. All special purpose register of the ATMEL are embedded into the data space, which has an address length of $m := 2$. As we focus on interrupt handling here, we will briefly describe the interrupt handling of the ATMEL and identify the special purpose registers within A that are essential for the interrupt handling.

The ATMEL has 21 different interrupts, each on an interrupt level of its own. The interrupts range over a non-maskable reset, externally generated interrupts, timer interrupts and internally generated interrupts caused, e.g., by completion of specific operations.

Each interrupt apart from the non-maskable reset can only occur when at least three conditions are met:

- interrupts are globally enabled, i.e., the *global interrupt enable* flag `I` in the status register `SREG` must be set,
- the enable interrupt bit of the specific interrupt must be set,
- the interrupt flag of the specific interrupt must be set.

The enable interrupt bit and the interrupt flag are found in different registers depending on the type of the interrupt. For timer interrupts, they are stored in the `TIMSK` and `TIFR` registers, for external interrupts in the `GICR` and the `GIFR` registers, and similar for the rest of the interrupts.

In order to model check real systems, where we have no control over the environment of our microcontroller, we must assume that input ports can have any value,

and that interrupts might occur at any time.

This is done by introducing nondeterminism for certain bits in our storage, namely for the input port bits of our system and the interrupt flags. The environment `env` is used to set these bits to $*$ before an instruction is carried out. Due to space limitations, we cannot give the full definition of the environment for the ATMEL, but we will illustrate it by an example with one of the built-in timers and with one of the external interrupts.

For the example, we use the function $nd : \mathbb{B}_* \to \mathbb{B}_*$ defined as $nd(0) := *, nd(1) = 1, nd(*) = *$. If an interrupt flag is already set, then this function leaves it set. If the interrupt flag is not set, or nondeterministic, then it could be set in this step, and thus must be made nondeterministic.

A timer interrupt for Timer 0 can only occur if the clock for Timer 0 is selected. The mode of the timer is set via the `CS02/CS01/CS00` bits of the `TCCR0` register.

The external interrupt 2 can only occur if a certain port is selected as the source for this interrupt. The port's setting is controlled via bit `DDB` in the port register `DDRB`. Thus we have:

$$\begin{aligned}&\texttt{TCCR0}[\texttt{CS02}] == 1 \vee \texttt{TCCR0}[\texttt{CS01}] == 1 \vee \texttt{TCCR0}[\texttt{CS00}] == 1 :\\ &\quad \texttt{TIFR}[\texttt{TOV0}] \leftarrow nd(\texttt{TIFR}[\texttt{TOV0}]);\\ &\texttt{DDRB}[\texttt{DDB2}] == 0 : \texttt{GIFR}[\texttt{INTF2}] \leftarrow nd(\texttt{GIFR}[\texttt{INTF2}]); \ldots\end{aligned}$$

As said before, the interrupt only occurs if the global interrupt enable bit in the status register, the bit in the interrupt mask and the interrupt flag are set. The following interrupt handler checks this first for the timer interrupt, then for the external interrupt.

$$\begin{aligned}&\texttt{SREG}[\texttt{I}] = 1 \wedge \texttt{TIMSK}[\texttt{TOIE0}] = 1 \wedge \texttt{TIFR}[\texttt{TOV0}] = 1 :, 18\downarrow\ >\\ &\texttt{SREG}[\texttt{I}] = 1 \wedge \texttt{GICR}[\texttt{INT2}] = 1 \wedge \texttt{GIFR}[\texttt{INTF2}] = 1 :, 36\downarrow > \ldots\end{aligned}$$

As can be seen, interrupt vectors are stored at memory locations 18 and 36 respectively, and the interrupt at 18 has priority over the other one.

5.5 The Bisimulation Proof

In order to reduce the effort in model checking which comes from introducing nondeterminism, we will show that for the interrupts, it is safe to make all interrupt bits below the current interrupt level nondeterministic. Assuming three interrupt levels stored in the lower three bits of the interrupt register, this means we can identify all states where these bits have the values $100, 110, 101, 111$, which is a reduction of a factor 4 for the interrupt handling. In the general, we gain a factor or 2^n for n interrupt levels, so for the ATMEL we gain a factor of up to 2^{20}.

To show this, we look at a fixed microcontroller $MC = (A, Q, \texttt{Ins}, \texttt{env}, \texttt{IH}, q_0, v_0)$. The interrupt handler and the environment handler are defined as sketched above, thus implementing the interrupt handling model of the ATMEL. The instruction space can hold an arbitrary program.

This MC induces a deterministic transition system $\rightarrow$ and a nondeterministic transition system $\rightarrow_{nd}$. Intuition tells us that the deterministic transition system

is the real behavior of the microprocessor when running the loaded program, and if the environment behaves as modeled. The nondeterministic transition system in contrast models how the simulation of the processor is implemented in [mc]square.

To show that the nondeterministic transition system is bisimilar to the deterministic one, formally we show that there is a bisimulation relation $\sigma : (Q \times V) \times (Q \times V_{nd})$ such that whenever $s \sigma s_{nd}$ and $s \to s'$, then there is s'_{nd} such that $s_{nd} \to_{nd} s'_{nd}$ and $s' \sigma s'_{nd}$, and vice versa.

The relation for which we show the result is just relating states which are more nondeterministic, i.e. for which there is a set $M \subseteq A_{bit}$ such that $w = \mathrm{ndet}(M, v)$. For those we claim that $(q, v)\sigma(q, w)$ is a bisimulation between (q, v) and (q, w).

So assume we have $(q, v) \to (q', v')$. Further let M such that $w = \mathrm{ndet}(M, v)$. If there is no interrupt involved, then we have $(q, w) \to_{nd} (q', w')$ for some M, N with $w = \mathrm{ndet}(M, v), w' = \mathrm{ndet}(N, v')$. By definition of σ, we have $(q', v')\sigma(q', w')$.

If an interrupt at level ℓ occurs, then we have $(q, w) \to_{nd} (q', w')$ for some M, N with $w = \mathrm{ndet}(M, v), w' = \mathrm{ndet}(\mathrm{IL}(\ell), \mathrm{ndet}(N, v'))$. Again, obviously we also have $(q', v')\sigma(q', w')$. A similar proof can be given for the reverse direction.

6 Case Study

This case study was first described in [15]. Here, we summarize the important details and reconsider the results under the aspect of DND for interrupts. In [15] only DND for values was considered. DND for interrupts was activated in all runs.

The case study was conducted on a laptop equipped with a Intel Core Duo CPU at 2.33 GHz, 4 GB main memory, and a hard disk with a capacity of 100 GB. [mc]square is completely written in Java, and hence, every operating system can be used. All programs used in this case study were developed by students during lab courses, exercises, diploma theses, or their working time. None of these programs was intentionally written to be model checked. All programs were run on the ATMEL ATmega16 microcontroller. Details about this case study can be found in [15].

As DND for interrupts cannot be deactivated in [mc]square because its operation is essential for the model checking of programs using interrupts, we can only show the differences obtained by DND of values. These three programs all use interrupts. Without using DND for interrupts the number of states would be considerably higher and model checking of these programs would not be possible.

Table 3 presents the outcome of this case study for the three programs. The first column shows the name of the program. In the second column it is indicated which abstraction techniques were used (here: *DND for interrupts only*, *DND for interrupts and for values*, and all abstraction techniques). The option all abstraction techniques includes DND, *path reduction*, and *dead variable reduction*. The column *# states stored* represents the number of different states stored in the state space. In contrast, the column *# states created* shows the number of all states created during building of the state space, including revisits. *Size [MB]* gives the size of the state space in main memory, and *Time [s]* shows the total time needed for building the state space including all preparatory steps (e.g., preprocessing, parsing, and static analyses) and model checking the formula **AG** true. We chose this formula

Program	Options used	# states stored	# states created	Size [MB]	Time [s]
plant	DND IR	801,616	854,203	240	23.19
	DND	188,404	195,955	57	4.39
	all	11,524	222,636	3.5	3.02
traffic light	DND IR	35,613	38,198	10	0.78
	DND	10,004	10,520	2.73	0.24
	all	523	13,069	0.21	0.17
window lift	DND IR	10,100,400	11,196,174	2,049	416.98
	DND	323,450	444,191	96	9.09
	all	10,699	463,129	3.26	7.43

Table 3
Effect of delayed nondeterminism on the state space size.

because it builds the complete state space, and it does not influence the abstraction techniques. In case a formula is chosen that makes an assumption about a certain memory location (e.g., register, I/O register, or variable), the abstraction techniques would no longer work on this memory location. As the formulas are different for each program, the influence on each program would be different. Therefore, a fair comparison of the state space sizes and the effect of the abstraction techniques would not be possible.

We can give some comments about the size of the state space when not using DND of interrupts. When using DND for interrupts, only enabled interrupts are fired by writing only possible value combinations into the flag registers. When not using DND for interrupts, all interrupts would be fired that have an active interrupt source by writing all value combinations into the flag registers.

The *plant* program consists of 225 lines of assembly code and uses two interrupts and one timer. The *traffic light* consists of 155 lines of assembly code and uses the same number of timers and interrupts as the *plant* program. The *window lift* has 289 lines of assembly code and uses again two interrupts and one timer. As all program use the same number of timers and interrupts, we only detail one of them.

The *plant* program uses one timer interrupt and one external interrupt. When using DND for interrupts, at most three combinations are written to the flag registers: timer interrupt occurred, external interrupt occurred, and no interrupt occurred. This is only done when the corresponding interrupts are enabled. When not using DND for interrupts more combination are written. In this case all interrupts are fired that have an active interrupt source. The interrupt source for the timer used in the *plant* program is actually the source for two different timer interrupts. The second timer interrupt is not used in this program but without using DND for interrupts, it would be fired. As all value combinations are written into the flag registers, at least nine combinations would be written. These nine combination would be created in every line of the program where the sources of the interrupts are active. The sources are active in almost all parts of the program including interrupt routines (in interrupt routines, other interrupts are usually deactivated). If nonde-

terminism of values is involved (e.g., input from the environment) additionally, the number of resulting states from these values would be multiplied with the number of interrupt value combinations. The same notes apply for the program *traffic light* and *window lift*.

DND has a major influence on the size of the state spaces. The influence of DND for values can be seen in Tab. 3 and is described in [15]. The influence of DND for interrupts cannot be seen in the table. Before we implemented DND for interrupts, we could hardly check a program using more than one interrupt. Now, we can usually check programs with up to five interrupts. DND for interrupts typically has a bigger influence on the size of the state space than DND for values has. In the previous paragraph, we gave an impression about the sizes of state spaces of these three programs when not using DND. [21] presents a case study where we model checked microcontroller programs, which were used to do a speed measuring for a model car. These programs used up to 5 interrupts and had up to 5000 lines of assembly code. Without using DND, [mc]square was not able to model check these programs. Using DND it was possible to model check them.

7 Conclusion & Future Work

In this paper delayed nondeterminism for interrupts, which is an abstraction technique implemented in [mc]square, was detailed and it was proven that DND for interrupts preserves a bisimulation relation. This is an important result as DND for interrupts cannot be deactivated by the user because this abstraction technique is too essential for [mc]square. Without this abstraction techniques, even small programs using more than one interrupt could not be model checked. As [mc]square is a CTL model checker, bisimulation is needed to preserve the validity of formulas. The DND of values preserves a simulation relation (see [15]) and hence, the validity of ACTL formulas is preserved. Nevertheless, DND of values can be deactivated by the user if the over-approximation is too coarse.

DND is an abstraction technique that introduces lazy states into [mc]square. A lazy state is a state that is mostly explicit but has symbolic parts. These symbolic parts remain symbolic until they are required in a computation. The moment they are required, they are lazily instantiated. Thereby, the approach used in [mc]square is no longer completely explicit but partly explicit and partly symbolic. DND has a significant influence on the size of state spaces. Without this abstraction technique, [mc]square could not model check most programs it can check using DND. As seen in Sect. 6 DND can be used together with other abstraction techniques implemented in [mc]square.

In the future, we want to investigate if we can establish a bisimulation relation for DND for values. The copying of values destroys the bisimulation relation. If we introduce instances of nondeterminism and copy these instances, instantiation such an instance would have an effect on all the instances and preserve the bisimulation. However, we have to observe the effects on the size of the state space and the number of different nondeterminism instances. Another thing that we want to implement is a model checking algorithm for a three-valued logic. This would make it possible to make propositions about registers used within the DND abstraction technique.

Summarizing, we think that this is a promising approach to analyze software for embedded systems. [mc]square can already handle programs of interesting size. Delayed nondeterminism is an abstraction technique that helps to tackle the state explosion problem. It can be combined with other techniques implemented in [mc]-square (e.g., path reduction and dead variable reduction). This technique can also be used for model checking software for many other microcontrollers. As we have experienced with delayed nondeterminism or path reduction (cf. [19]), there are abstraction techniques which perform better when model checking assembly code. Hence, we will focus future research on domain specific abstraction techniques.

References

[1] Balakrishnan, G., T. Reps, D. Melski and T. Teitelbaum, *WYSINWYX: What you see is not what you execute*, in: *Verified Software: Theories, Tools, Experiments* (2007), to appear.

[2] Bryant, R. E., *A methodology for hardware verification based on logic simulation*, Journal of the ACM **38** (1991), pp. 299–328.

[3] Clark, A., *A lazy non–deterministic functional language* (2000), http://www.dcs.kcl.ac.uk/staff/tony/docs/LazyNonDetLanguage.ps.

[4] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, Cambridge, Massachusetts, 1999.

[5] Emerson, E. A., *Temporal and modal logics*, in: *Handbook of Theoretical Computer Science, vol. B*, Elsevier, 1990 .

[6] Godefroid, P., N. Klarlund and K. Sen, *DART: directed automated random testing*, SIGPLAN Not. **40** (2005), pp. 213–223.

[7] Heljanko, K., *Model checking the branching time temporal logic CTL*, Research Report A45, Helsinki University of Technology (1997).

[8] Holzmann, G. J., "The Spin Model Checker: Primer and Reference Manual," Addison-Wesley, 2003.

[9] Larsen, K. G., P. Pettersson and W. Yi, Uppaal *in a Nutshell*, Int. Journal on Software Tools for Technology Transfer **1** (1997), pp. 134–152.

[10] Launchbury, J., *An natural semantics for lazy evaluation*, in: *Proc. 20th ACM Symp. on Principles of Programming Languages (POPL '93)* (1993), pp. 144–154.

[11] Leven, P., T. Mehler and S. Edelkamp, *Directed error detection in C++ with the assembly-level model checker StEAM*, in: *Model Checking Software (SPIN)*, Lecture Notes in Computer Science **2989** (2004), pp. 39–56.

[12] Mehler, T., "Challenges and Applications of Assembly-Level Software Model Checking," Ph.D. thesis, Universität Dortmund (2005).

[13] Mercer, E. G. and M. D. Jones, *Model checking machine code with the GNU debugger*, in: *SPIN Workshop on Model Checking of Software*, LNCS **3639** (2005), pp. 251–265.

[14] Meseguer, J. and P. Thati, *Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols*, ENTCS **117** (2005), pp. 153–182.

[15] Noll, T. and B. Schlich, *Delayed nondeterminism in model checking embedded systems assembly code*, in: *Proc. of 3rd Annual Haifa Verification Conf. (HVC 2007)*, LNCS (2007), to appear.

[16] Schlich, B. and S. Kowalewski, *Model checking C source code for embedded systems*, in: T. Margaria, B. Steffen and M. G. Hinchey, editors, *Proc. IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (IEEE/NASA ISoLA 2005)*, NASA/CP-2005-212788 (2005), pp. 65–77.
URL http://www-i11.informatik.rwth-aachen.de/schlich.html

[17] Schlich, B. and S. Kowalewski, *[mc]square: A model checker for microcontroller code*, in: T. Margaria, A. Philippou and B. Steffen, editors, *Proc. 2nd Int'l Symp. Leveraging Applications of Formal Methods, Verification and Validation (IEEE-ISoLA 2006)*, 2006, to appear in: IEEE proceedings.

[18] Schlich, B. and S. Kowalewski, *An extendable architecture for model checking hardware-specific automotive microcontroller code*, in: E. Schnieder and G. Tarnai, editors, *Proc. 6th Symp. Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)* (2007), pp. 202–212.

[19] Schlich, B., J. Löll and S. Kowalewski, *Application of static analyses for state space reduction to microcontroller assembly code*, in: *Proc. 12th Int'l Workshop Formal Methods for Industrial Critical Systems (FMICS 2007)*, LNCS (2007), to appear.

[20] Schlich, B., M. Rohrbach, M. Weber and S. Kowalewski, *Model checking software for microcontrollers*, Technical Report AIB-2006-11, RWTH Aachen University (2006).
URL http://aib.informatik.rwth-aachen.de/2006/2006-11.pdf

[21] Schlich, B., F. Salewski and S. Kowalewski, *Applying model checking to an automotive microcontroller application*, in: *Proc. IEEE 2nd Int'l Symp. Industrial Embedded Systems (SIES 2007)* (2007), to appear.

[22] Sen, K., D. Marinov and G. Agha, *CUTE: a concolic unit testing engine for C*, in: *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), pp. 263–272.

[23] Vergauwen, B. and J. Lewi, *A linear local model checking algorithm for CTL*, in: *Proc. 4th Int'l Conf. on Concurrency Theory (CONCUR '93)*, Lecture Notes in Computer Science **715** (1993), pp. 447–461.

[24] Visser, W., K. Havelund, G. Brat, S. Park and F. Lerda, *Model checking programs*, Automated Software Engineering Journal **10** (2003).

CVM - A Verified Framework for Microkernel Programmers

Tom In der Rieden[1]

Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI)
Saarbrücken, Germany

Alexandra Tsyban[1]

Computer Science Dept.
Universität des Saarlandes
Saarbrücken, Germany

Abstract

CVM (communicating virtual machines) is a computational model for concurrent user processes interacting with a generic microkernel—supporting virtual memory—and devices. In this paper, we introduce the computational models needed to define CVM. Furthermore, we describe how CVM can be implemented by means of a concrete kernel, thus providing a trustworthy platform for microkernel programmers. Last but not least, we give an overview on the model formalization and implementation correctness proof, which has been conducted in the interactive theorem prover Isabelle for the most part. An endeavor like this is tedious and of a considerable complexity. Thus, we do not try to present all details, but provide references to publications covering specific aspects.

Keywords: Operating Systems, Microkernel, Systems Verification, Isabelle, Theorem Proving

1 Introduction

Operating systems are crucial components in nearly every computer system. They provide plenties of services and functionalities, e.g. managing inter-process communication, device access, and memory management. Obviously, they play a key role in the reliability of such systems and in fact, a considerable share of hacker attacks target operating system vulnerabilities. Thus, proving a computer system to be safe and secure requires to prove its operating system to be safe and secure.

At first sight, this appears to be a mission impossible because of the sheer size of operating system implementations. For example, the Linux 2.6.0 kernel released

[1] This work was partially funded by the German Federal Ministry of Education and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS 07008. The responsibility for this article lies with the authors.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

in late 2003 has nearly 6 million lines of code. Yet, the idea of having a small and reliable kernel is not new and has led to the development of so-called 2nd generation microkernels like L4 [16]. Microkernels offer elementary, but sufficient functionality, and can therefore be of relatively small size. For instance by using them as a trusted platform, we can run two operating systems on top of it, one small and reliable for critical applications, and a conventional one for all other tasks [24].

In this paper, we describe how a whole framework, featuring virtual memory support, memory management, system calls, user defined interrupts, etc.—thus providing a trustworthy platform for microkernel programmers—can be proven correct. We introduce a computational model called CVM (communicating virtual machines), that formalizes concurrent user processes interacting with a generic (abstract) microkernel and devices. To establish interaction, the abstract kernel features special functions called CVM primitives, which are invoked by the user processes and alter process or device configurations, e.g. by copying data from one process to another. By linking a CVM implementation to an abstract kernel, we obtain a concrete kernel ('personality').

For each layer in the computer system—hardware, devices, user processes, and abstract kernel—we define a formal model. Implementation correctness is defined by several simulation relations between these layers. The proofs are conducted in the interactive theorem prover Isabelle/HOL [22] and have already been completed to a large extent.

CVM is used in the Verisoft project [29] in two personalities: (i) VAMOS is a microkernel used in an academic stack, where on top of it a simple operating system (SOS) is running, and (ii) OLOS, an OSEKtime-like operating system, is used in a distributed automotive real-time system establishing eCall functionality [13].

The remainder of this paper is structured as follows. In Sect 2 we list some related work. Sect. 3 introduces some notation needed in Sect. 4 to define our models formally. We present a generic framework for devices in Sect. 4.1. In particular, we show how physical machines and external devices can be coupled formally (Sect. 4.2). User processes are modeled by assembler machines running on virtual memory (Sect. 4.3), while computations of the abstract kernel are defined by *C0* semantics (Sect. 4.4). In Sect. 5 we sketch the construction of the concrete kernel containing the CVM implementation. The simulation relations that establish CVM implementation correctness are described in Sect. 6. The status quo of the formal verification is presented in detail in Sect. 7. We conclude in Sect. 8.

2 Related Work

First attempts to use theorem provers to specify and even prove correct operating systems were made as early as the seventies in PSOS [20] and UCLA Secure Unix [32]. However a missing—or to a large extend underdeveloped—tool environment made mechanized verification futile. With the CLI stack [4], a new pioneering approach for pervasive system verification was undertaken. Most notably, the simple kernel KIT was developed and its machine code implementation was proven to be correct. Compared to modern kernels KIT was very limited, in particular, it lacked interaction with devices. The project L4.verified [9] focuses on the verification of an

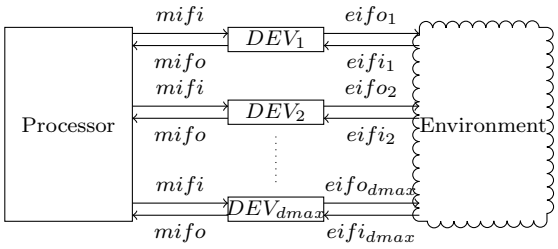

Fig. 1. Device Model

efficient microkernel, rather than on formal pervasiveness, as no compiler correctness or accurate device interaction is considered. The microkernel is implemented in a larger subset of C, including pointer arithmetic and an explicit low-level memory model [31]. However with inline assembler code we gain an even more expressive semantics as machine registers become visible if necessary. So far, only exemplary portions of kernel code were reported to be verified, the virtual memory subsystem uses no demand paging [30]. For code verification L4.verified relies on the Verisoft's Hoare environment [26]. In the FLINT project, an assembly code verification framework is developed and code for context switching on a x86 architecture was formally proven [21]. Although a verification logic for assembler code is presented, no integration of results into high-level programming languages is undertaken. The VFiasco project [12] aims at the verification of the microkernel Fiasco implemented in a subset of C++. Code verification is performed in a embedding of C++ in PVS and there is no attempt to map the results down to the machine level.

3 Notation

We use $f : A \rightharpoonup B$ to denote a partial mapping f from sets A to B, while $g : A \to B$ stands for a total mapping. We denote the concatenation of bit strings $a \in \{0,1\}^n$ and $b \in \{0,1\}^n$ by $a \circ b$. For bits $x \in \{0,1\}$ and positive natural numbers $n \in \mathbb{N}^+$, we define inductively $x^1 = x$ and $x^n = x^{n-1} \circ x$, e.g. $0^5 = 00000$ and $1^2 = 11$. For $x \in \{0,1\}^n$, $x[i], 0 \leq i < n$ denotes the bit at position i of the bit string. $\mathbb{N}_i$ with $i \in \mathbb{N}^+$ denotes the natural interval $[0, i-1]$. For finite sequences $seq : \mathbb{N} \rightharpoonup T$, we use shorthand notation $seq = hd; tl$ where hd denotes the head of the sequence, i.e. the element $seq[0]$, and tl the remaining elements.

4 Computational Models

We have developed a generalized framework to model different devices in a uniform way (Sec. 4.1). Physical machines (Sec. 4.2) are used to specify the underlying microprocessor hardware. User process computations are modeled by assembler semantics (Sec. 4.3), abstract kernel computations by *C0* semantics (Sec. 4.4). Finally, we combine the above computational models to specify CVM (Sect. 4.5).

4.1 Devices

We use devices in two ways in CVM. First, the page fault handler described in Sect. 5.3 uses a hard disk as swap device. Second, we offer a range of typical

devices accessible by user processes through special kernel calls. Currently, we support models for five different device types: (i) a hard disk, e.g. used as the swap device for memory virtualization [11], (ii) a timer, which can be used for scheduling user processes, (iii) a network interface, (iv) an UART serial interface, which can be used to set up a terminal [1], and (v) an automotive bus controller, which is used in Verisoft's Automotive subproject [14,13]. Device communication happens on many different layers throughout our stack. Nevertheless, in order to establish a uniform way of interacting with devices, we have developed a generic device framework featuring standardized transition functions for all layers.

For each device type, we define a specific device configuration, e.g. c_{hd} for hard disks. The set of all specific device configurations (including a generic error state $\perp$) is denoted by $Conf_{\text{sdevs}}$. Furthermore, we define the set of device IDs by $DID = \mathbb{N}^{+}_{dmax+1}$, whereas $dmax$ is fixed and determines the maximal number of devices. Formally, the generalized device configuration is defined by the mapping $Conf_{\text{devs}} : DID \to Conf_{\text{sdevs}}$.

Devices are memory-mapped and can communicate in two directions, namely with the processor and with the environment (e.g. with a user or a screen). Thus, we define two transition functions, one for internal and one for external communication. The generalized internal transition function is parameterized over inputs from the processor ($Mifi$). One element of the processor input is defined by a tuple $mifi = (id, rd, wr, ad, count, data)$, whereas (i) $id \in DID$ denotes the device ID, (ii) $rd \in \{0,1\}$ denotes the read flag, (iii) $wr \in \{0,1\}$ denotes the write flag, (iv) ad gives the device port where to read from or where to write to, and (v) $data$ finally specifies the data to be written if $wr = 1$.

Furthermore, internal steps do not only yield a successor device configuration, but also output to the processor ($Mifo \subseteq \mathbb{N}$) and, potentially, to the environment, which is specific for each device type: $Eifo = \{Eifo_{\text{hd}}, Eifo_{\text{abc}}, \ldots\}$. Now we can formally define the generalized internal transition function $\delta_{\text{dint}} : Conf_{\text{devs}} \times Mifi \to Conf_{\text{devs}} \times Mifo \times Eifo$.

Input from the environment is also device type specific; We specify the generalized set of environment input by $Eifi = \{Eifi_{\text{hd}}, Eifi_{\text{abc}}, \ldots\}$. Again, devices may generate output. The external transition function is given by $\delta_{\text{dext}} : Conf_{\text{devs}} \times DID \times Eifi \to Conf_{\text{devs}} \times Eifo$.

Of course this way of dealing with devices is not the only possible on; modern architectures mostly rely on devices with direct-memory access (DMA). Yet, this makes modeling much more difficult and would require considerable changes on the hardware implementation. First, either the processor would not be the sole bus master any longer, but I/O MMUs would have to guard the bus, or one would have to trust devices not to access sensitive data; second, DMA regions would have to be excluded from caching. Since these regions are set up dynamically, this would require hardware support for explicit cache flushing.

4.2 Physical machines and instruction set architecture

The lowest layer in our stack is given by the architecture of the underlying VAMP microprocessor [6]. The VAMP provides a single-level address translation mecha-

nism [8,10] and supports memory-mapped devices [1,11].

In order to realize memory virtualization, the VAMP runs in two modes: user mode and system mode. In user mode, all addresses are virtual and have to be translated first before accessing memory [7]. However, in system mode, we deal with physical addresses that can be used without translation. In our scenario, the microkernel runs in system mode while the user processes run in user mode.

The processor and the devices may either progress individually or communicate with each other. Communication is established either by the processor executing a memory operation to a special memory region assigned to devices (see Fig. 3) or by the device causing an external interrupt.

Physical machines are the hardware model for a system programmer. Due to space limitations, we will only introduce the relevant parts here.

A physical machine configuration c_{phys} comprises the registers, the program counters, and the memory content. The register file is split into two parts: (i) $gpr : \{0,1\}^5 \to \{0,1\}^{32}$, the general purpose register file, and (ii) $spr : \{0,1\}^5 \rightharpoonup \{0,1\}^{32}$, the special purpose register file. For shorthand notation, we use symbolic names for special purpose registers, e.g. $Edata$ for $spr(00101)$. In order to implement the delayed branch mechanism as described in [17], we specify a program counter $pc \in \{0,1\}^{32}$ and a delayed program counter $dpc \in \{0,1\}^{32}$. Finally, there is a word-addressed physical memory $pm : \{0,1\}^{30} \rightharpoonup \{0,1\}^{32}$. We formally denote a physical machine configuration by the tuple $c_{\text{phys}} = (gpr, spr, pcp, dpc, pm)$.

An instruction set architecture (ISA) is given by a transition function δ_{phys}, that maps a configuration c_{phys} and external event signals $eev \in \{0,1\}^{dmax}$ to a next configuration $c'_{\text{phys}} = \delta_{\text{phys}}(c_{\text{phys}}, eev)$. Due to space limitations, we will not give a full formal definition of the ISA transition function, but only an idea of it.

The transition function depends on the special purpose register $mode$, where $mode = 0$ denotes system mode and $mode = 1$ denotes user mode. For system mode, the transition function is simply defined by the instruction to which $c_{\text{phys}}.dpc$ points (see [5,17]). In user mode, memory accesses are subject to address translation: they either cause a page fault or are redirected to the translated physical address $pma(c_{\text{phys}}, va)$ for a given virtual address va. For details on VAMP address translation see [8].

In order to define $\delta_{\text{phys}}(c_{\text{phys}}, eev)$ more formally, we need some helper functions: (i) $I(c_{\text{phys}}) = c_{\text{phys}}.pm(c_{\text{phys}}.dpc)$ denotes the instruction to be executed in configuration c_{phys}, (ii) predicates $?lw(c_{\text{phys}})$ and $?sw(c_{\text{phys}})$ distinguish $I(c_{\text{phys}})$ being a 'load word' or a 'store word' instruction, (iii) $RS1(c_{\text{phys}})$, $RS2(c_{\text{phys}})$, and $RD(c_{\text{phys}})$ are returning the general purpose register operands of $I(c_{\text{phys}})$, (iv) $imm(c_{\text{phys}})$ returns the immediate constant of $I(c_{\text{phys}})$, and (v) $ea(c_{\text{phys}}) = c_{\text{phys}}.gpr(RS1(c_{\text{phys}})) + imm(c_{\text{phys}})$ specifies the effective address of $I(c_{\text{phys}})$, i.e. its memory operand.

Note, that for CVM, only the kernel running in system mode interacts directly with devices, thus address translation does not affect. Devices are memory mapped, i.e. a part of the memory is shared by both the processor and the devices (cf. Fig. 3). We denote the set of addresses in this part of the memory by DIO. Let the predicate $?int(i)$ denote, if the device with ID i is in an interrupt state. We can now define the external event signals as $eev = ?int(dmax) \circ ?int(dmax-1) \circ \ldots \circ ?int(1)$.

Let us introduce an input alphabet $\mathbb{I} = (DID \times Eifi) \cup \{0\}$ and an output alphabet $\mathbb{O} = Eifo \cup \{\varepsilon\}$. Formally the combined transition function of physical machines and devices $\delta_{p\&d}(c_{\text{phys}}, c_{\text{devs}}, in) = (c'_{\text{phys}}, c'_{\text{devs}}, out)$ is defined for $in \in \mathbb{I}$ and $out \in \mathbb{O}$ as follows. For external device input ($in \neq 0$), we execute the external device transition function, thus $c'_{\text{phys}} = c_{\text{phys}}$ and $(c'_{\text{devs}}, out) = \delta_{\text{dext}}(c_{\text{devs}}, in)$. For processor steps ($in = 0$), we distinguish between steps with device interaction, i.e. $ea(c_{\text{phys}}) \in DIO$, and without device interaction: (i) in the former case, we execute both the processor and the internal device transition function: $c'_{\text{phys}} = \delta_{\text{phys}}(c_{\text{phys}}, eev)$ and $(c'_{\text{devs}}, mifo, out) = \delta_{\text{dint}}(c_{\text{devs}}, mifi)$. If $?lw(c_{\text{phys}})$, we set $c'_{\text{phys}}.gpr(RD(c_{\text{phys}})) = mifo$, otherwise we discard $mifo$. (ii) In the latter case, we execute the transition function of the processor: $c'_{\text{phys}} = \delta_{\text{phys}}(c_{\text{phys}}, eev)$ and $(c'_{\text{devs}}, out) = (c_{\text{devs}}, \varepsilon)$. Here, $mifi$ is obtained by a helper function $dec(ea(c_{\text{phys}})) = (i, ad)$ returning the device id i and port ad for a given effective address, such that $mifi = (i, ad, ?lw(c_{\text{phys}}), ?sw(c_{\text{phys}}), c_{\text{phys}}.gpr(RD(c_{\text{phys}})))$.

The n-step transition function $\delta^n_{p\&d}$ takes initial configurations c_{phys} and c_{devs}, and a input sequence ins of length m, with elements $m_i \in \mathbb{I}$ and $m \geq n$. While executing single steps, $\delta^n_{p\&d}$ generates an output sequence $outs$ of length m and elements in $\mathbb{O}$. We define $\delta^n_{p\&d}$ recursively: (i) $\delta^0_{p\&d}(c_{\text{phys}}, c_{\text{devs}}, ins) = (c_{\text{phys}}, c_{\text{devs}}, \varepsilon)$, and (ii) $\delta^{i+1}_{p\&d}(c_{\text{phys}}, c_{\text{devs}}, ins) = \delta_{p\&d}(c'_{\text{phys}}, c'_{\text{devs}}, ins(i+1))$ with $\delta^i_{p\&d}(c_{\text{phys}}, c_{\text{devs}}, ins) = (c'_{\text{phys}}, c'_{\text{devs}}, outs)$

4.3 Assembler Semantics

User processes are applications running on top of the microkernel. Given that we also want to consider malevolent (hacker) applications, we restrain from any programming restrictions imposed by C and model all processes as assembler machines. More precisely, since the microkernel is providing memory virtualization and these applications run on a uniform virtual memory, we will use virtual assembler machines. A virtual machine configuration c_{ASM} is closely related to the physical machine configuration describing the hardware. It still comprises the register files and the program counters. We consider register numbers as naturals and their contents as integers. Furthermore, only a subset of the special purpose registers available in the real hardware is visible here and the instruction set is limited.

We formally define the register files (similar to Sect. 4.2) as (partial) mappings, i.e. $gpr : \mathbb{N}_{32} \to \mathbb{Z}$ and $spr : \mathbb{N}_{32} \rightharpoonup \mathbb{Z}$, and the two program counters $dpc, pcp \in \mathbb{N}$. The memory is given by $mm : \mathbb{N} \rightharpoonup \mathbb{Z}$, such that the overall assembler configuration is specified by the tuple $c_{\text{ASM}} = (gpr, spr, pcp, dpc, mm)$.

The transition function, which maps a given assembler configuration c_{ASM} either to its successor configuration or to an error state $\perp$: $\delta_{\text{ASM}} : Conf_{\text{ASM}} \to Conf_{\text{ASM}} \cup \{\perp\}$, is defined over the current instruction to which $(dpc.mm(c.dpc))$ points. For example, the error state can be reached if the user process tries to access a restricted special purpose register.

Let $\delta^n_{\text{ASM}} : Conf_{\text{ASM}} \to Conf_{\text{ASM}} \cup \{\perp\}$ denote the function that applies the transition function $n \in \mathbb{N}$ times. We define inductively (i) $\delta^0_{\text{ASM}}(c_{\text{ASM}}) = c_{\text{ASM}}$ and (ii) $\delta^{i+1}_{\text{ASM}}(c_{\text{ASM}}) = \delta_{\text{ASM}}(c'_{\text{ASM}})$ if $\delta^i_{\text{ASM}}(c_{\text{ASM}}) = c'_{\text{ASM}}$ and $c'_{\text{ASM}} \neq \perp$, else $\perp$.

4.4 C0 Small Step Semantics

C0 is the C-like imperative programming language developed and widely used in Verisoft. It features sufficient functionality to implement system software and applications, yet having a concise formal semantics which allows for the—more or less—efficient verification of code with several thousands lines, e.g. a non-optimizing compiler and a simple email client [23,3].

A *C0* program is identified by its functions—including information about their list of parameters and local variables—, the type name environment and the list of global variables. *C0* supports four elementary types ($Bool$, $Integer$, $Unsigned$ and $Char$) and allows for non-elementary, recursive data types: $Arr(l,t)$ denotes the array with l elements of type t and, for types t_i and component names n_i, $Struct([(n_0,t_0),\ldots,(n_l,t_{l-1})])$ denotes a structure type with l components. *C0* pointers are denoted by $Ptr(tn)$ where tn stands for a type name defined in the type name environment $tenv$, a mapping $tenv : \Sigma^+ \rightharpoonup ty$ mapping type names to types. The procedure table contains the information about all functions of a *C0* program. Formally, it is a partial mapping $ptable : \Sigma^+ \rightharpoonup fdesc$ of function names to their corresponding descriptors, containing information on function body, parameters, local variables and return type. The global variables are defined by a sequence of variable names and their associated types: $st : \mathbb{N} \rightharpoonup \Sigma^+ \times ty$, called symbol table. We will not discuss in detail *C0* statements ($Stmt$) and expressions ($Expr$). The definitions of both are straightforward and are presented exhaustively in [15].

A *C0* configuration $c_{C0} = (mem, pr)$ consists of the memory configuration mem—storing information about the (possibly dynamically allocated) program variables and their values—and the program rest pr. Variables are represented in a generalized way as so-called g-vars, defined inductively as: a global variable of name x as $gvar_{gm}(x)$, a local variable of name x in the i-th stack frame as $gvar_{lm}(i,x)$, and a nameless heap variable with index i as $gvar_{hm}(i)$. If s is a g-var of structural type, then its component with name cn is also a g-var: $gvar(s,cn)$. Similar, for a g-var a of array type, its i-th element is also a g-var: $gvar(a,i)$. A memory configuration is given by a triple consisting of (i) a global memory frame $mem.gm : mframe$, (ii) a local memory stack $mem.lm : \mathbb{N} \rightharpoonup mframe \times gvar$ [2], and (iii) a heap memory frame $mem.hm : mframe$. Each frame contains a symbol table and a content $ct : \mathbb{N} \rightharpoonup mcell$, mapping addresses to typed memory cells. Memory cells can store the value of an elementary type variable, whereas pointers are represented by a g-var or the null pointer value $Null$; values of aggregate variables are stored in consecutive memory cells. The second component of a *C0* configuration is the program rest, a sequence of statements still to be executed: $pr = s_1,\ldots,s_n$ with $s_i \in Stmt$.

Given a type name environment te and a procedure table pt, the transition function maps the current *C0* configuration either to its successor configuration or to an error state $\perp$: $\delta_{C0} : tenv \times ptable \times Conf_{C0} \rightarrow Conf_{C0} \cup \{\perp\}$. δ_{C0} is defined inductively over the program rest (see [15] for a detailed definition).

We define δ^n_{C0}, which executes the transition function n times, by induction on n:

[2] local memory frames have an additional g-var defining the memory location where the function return value is to be stored

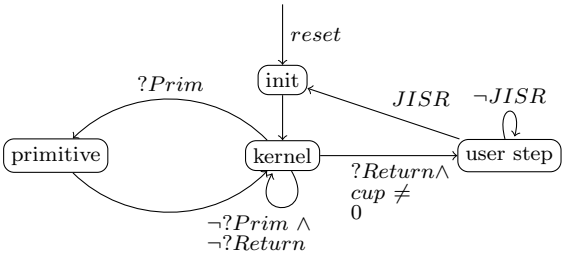

Fig. 2. CVM Control Flow

(i) $\delta^0_{C0}(te, pt, c_{C0}) = c_{C0}$ (ii) $\delta^{i+1}_{C0}(te, pt, c_{C0}) = \delta_{C0}(te, pt, c'_{C0})$, if $\delta^i_{C0}(te, pt, c_{C0}) = c'_{C0}$ and $c'_{C0} \neq \perp$, else $\perp$.

4.5 CVM Semantics

Communicating virtual machines (CVM) are a computational model for a fixed number of processes. The processes can interact with each other and with a fixed number of devices, whereas all communication is handled by a generic abstract microkernel offering various specific kernel calls (CVM primitives). So far, there is no support for shared memory, neither between devices and processes nor between processes themselves. We use assembler semantics as introduced in Sect. 4.3 to model user process computations and the *C0* semantics from Sect. 4.4 for the computations of the abstract kernel. Device behavior is defined by the semantics described in Sect. 4.1.

A CVM configuration $c_{\text{CVM}} = (kernel, proc, devs, sr, cup)$ comprises the following five components: (i) Let $PID = \mathbb{N}^+_{pmax+1}$ denote the set of user process IDs with a fixed $pmax$. Then, a user processes configuration $procs$ is formally a mapping of process IDs to assembler configurations: $procs : PID \to Conf_{ASM}$. (ii) The $kernel$ part is specified by a type name environment, a procedure table, and a *C0* configuration: $kernel = (tenv, pt, conf)$. (iii) The device configuration is given by a generalized device configuration $devs$. (iv) $cup \in PID \cup \{0\}$ specifies the current process ID or, in case of $cup = 0$, the kernel. (v) $sr \in \{0,1\}^{dmax-1}$, the interrupt mask for the devices. If $sr[i] = 0$, then interrupts of the device with ID $did = i + 2$ are masked [3].

In each CVM step, either the kernel, or one user process, or one device progresses. One step in a CVM computation is defined by the transition function $\delta_{\text{CVM}} : Conf_{CVM} \times \mathbb{I} \to (Conf_{CVM} \cup \{\perp\}) \times \mathbb{O}$, where $\perp$ denotes the error state. In the following definitions, we only mention components that are changing in one step of the computation.

Given a CVM configuration c_{CVM} and a parameter in: if $in \neq 0$, we execute the external transition function of the device part of the CVM configuration $c_{\text{CVM}}.devs$ as described in Sect. 4.1: $\delta_{\text{dext}}(c_{\text{CVM}}.devs, in) = (c'_{\text{CVM}}.devs, out)$.

For $in = 0$, execution depends on the value of $c_{\text{CVM}}.cup$: if $c_{\text{CVM}}.cup = i > 0$, the user process $c_{\text{CVM}}.procs(i)$ makes a step, otherwise the kernel progresses.

Let the predicate $JISR(c_{\text{ASM}}, c_{\text{devs}}) \in \{0,1\}$ denote, that an interrupt occurred in the current user process configuration c_{ASM} and device configuration c_{devs} w.r.t.

[3] Note, that the hard disk used for swapping (device ID $did_{shd} = 1$) is not visible in the CVM specification.

the interrupt mask $c_{\text{CVM}}.sr$. Then we compute the (masked) exception cause $mca(c_{\text{ASM}}) \in \mathbb{N}$ and a potential parameter $edata(c_{\text{ASM}}) \in \mathbb{N}$ (e.g. in case of a `trap` exception). Details on *JISR*, *mca*, and *edata* can be found in [8,17]. For $\neg JISR(c_{\text{CVM}}.procs(c_{\text{CVM}}.cup), c_{\text{CVM}}.devs)$, we execute the transition function δ_{ASM} as described in Sect. 4.3: $c'_{\text{CVM}}.procs(c_{\text{CVM}}.cup) = \delta_{\text{ASM}}(c_{\text{CVM}}.procs(c_{\text{CVM}}.cup))$. Otherwise, a visible interrupt has occurred and kernel execution starts at the entry point given by the *C0* function *kdispatch* with parameters *mca* and *edata*. We set $c_{\text{CVM}}.cup = 0$ and the kernel's program rest to the function call $c'_{\text{CVM}}.kernel.conf.pr = SCall(kret, kdispatch, mca(c_{\text{ASM}}), edata(c_{\text{ASM}}))$ where *kret* denotes the return variable and *SCall* is the *C0* function call statement.

After booting and after an interrupt, kernel execution starts by calling the function *kdispatch*. Note, that while the kernel runs, interrupts are disabled, i.e. our kernel is non-interruptible ('non-preemptive'). If $c_{\text{CVM}}.cup = 0$ and the kernel program rest does not start with a function call to a CVM primitive , we simply execute the *C0* transition function as described in Sect. 4.4: $c'_{\text{CVM}}.kernel = \delta_{C0}(c_{\text{CVM}}.kernel)$. Otherwise, we have $c_{\text{CVM}}.kernel.conf.pr = ESCall(v, prim, expr_1, \ldots, expr_n); r$ for a CVM primitive *prim*, an integer return variable v and unsigned expressions $expr_1, \ldots, expr_n \in Expr$. Here, *ESCall* is the *C0* statement for external function calls, i.e. functions with declarations but without a body (Sect. 5 explains how to get a fully implemented kernel). Each primitive *prim* is specified by a function $prim_S$, which takes n natural arguments and a CVM configuration c_{CVM}, returning an updated CVM configuration c'_{CVM} or the error state $\perp$. Let $eval_r$ be the evaluation function for righthand side *C0* expressions as defined in [15]. Then, we compute for $1 \leq i \leq n$: $val_i = eval_r(c_{\text{CVM}}.kernel, e_i)$ and set $(c'_{\text{CVM}}) = prim_S(c_{\text{CVM}}, val_1, \ldots, val_n)$.

For example, CVM provides primitives (i) *Reset* and *Clone* for process initialization, (ii) *Alloc* and *Free* for increasing and decreasing memory of an user process, (iii) *Copy* to copy data from one process to another, (iv) *GetGPR* and *SetGPR* to read and write registers of user processes, (v) *GetWord* and *SetWord* to read from and write to a user process memory address, (vi) *InWord* and *OutWord* for device communication, and (vii) *SetMask* for setting the CVM interrupt mask. For a full list of primitives see [27].

Due to lack of space, we exemplify by the specification of *SetGPR*: Given a configuration with kernel program rest $ESCall(v, SetGPR, expr_1, expr_2, \ldots, expr_5); r$. We set $SetGPR_S(c_{\text{CVM}}, pid, i, y) = c'_{\text{CVM}}$ with (i) $c'_{\text{CVM}}.proc(pid).gpr(i) = y$, (ii) $c'_{\text{CVM}}.kernel.conf.pr = r$, and (iii) $c'_{\text{CVM}}.kernel.conf.mem = mem_update(c_{\text{CVM}}.kernel.conf.mem, v, 0)$ with the *C0* memory update function as defined in [15].

Note, that since the whole stack runs on one single processor, it is legal to assume that either the kernel or an user process perform a step. This is not that obvious for the devices. Remember that devices are memory-mapped and access to these memory regions happens only within the dedicated primitives of the kernel. These primitives are written in assembler, hence steps in the kernel and on the physical machine have the same granularity: one instruction. Additionally, the actual synchronization with the device can be mapped down to one single instruction, namely a load word or store word instruction. All other steps during kernel execution are

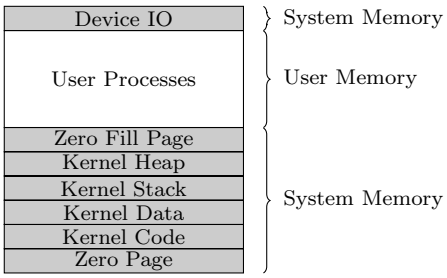

Fig. 3. CVM Memory Map

independent from the device computation. This means, all interleavings possible between physical machine steps and devices—as seen in Sect. 4.2—are also possible on between the kernel and the devices.

5 CVM Implementation

In this section, we will give an overview on the CVM implementation details and how to merge such an implementation with the abstract kernel in order to obtain a compilable and thus executable kernel.

5.1 Data Structures

To simulate virtual machines and multi-processing, the CVM implementation has to maintain certain data structures: (i) in the kernel global memory (kernel data), we store an array of process control blocks $pcb[i], i \in PID$ for all user processes. One process control block has components $pcb[i].r$ for each register and the program counters of c_{phys}, (ii) the global memory variable *cup* keeps track of the current user process as specified in $c_{\text{CVM}}.cup$ and similarly, *sr* for $c_{\text{CVM}}.sr$, (iii) in the global memory variable *kheap*, we store the end address of the kernel heap, (iv) the array *ptspace* on the kernel heap holds the page tables of all user processes, and (v) data structures of the page-fault handler (see Sect.5.3) necessary for the management of physical and swap memory.

5.2 Entering and Leaving System Mode

Whenever we enter system mode, i.e. the kernel starts to execute, we initialize its program rest with $c_{\text{CVM}}.kernel.conf.prog = init$. In all cases but reset, *init* will take the current process and store its registers into the corresponding control block $pcb[cup]$. Then, the kernel is initialized and the CVM dispatcher *cvmdispatch* is called with parameters $pcb[cup].eca$, $pcb[cup].edata$, and $pcb[cup].edpc$. As mentioned in Sect.4.5, interrupts are to be invisible in system mode. We achieve this by zeroing the status register $c_{\text{phys}}.SR$. In case of a page fault, the page fault handler is invoked by *cvmdispatch*. Otherwise we continue with a call to the abstract kernel dispatcher *kdispatch* with parameters $pcb[cup].eca$ and $pcb[cup].edata$.

To leave system mode and with $i \in PID$ being the user process to be started, we set $cup = i$ and restore the process from its control block $pcb[i]$. Finally, we leave kernel execution with a return from exception instruction (`rfe`).

Note, that a scheduler is not part of the concrete kernel, i.e. the abstract kernel has to take care of handling timer interrupts. Thus, we are not giving any guarantees

5.3 Page Fault Handler

For $pcb[cup].eca = 8$, the user process with ID *cup* has caused a page fault on fetch interrupt (*pff*), i.e. the process' delayed pc points to an address not present in physical memory. Thus, the page fault handler is called with parameters *cup* and $pcb[cup].edpc$. For $pcb[cup].eca = 16$, we are dealing with a page fault on load/store (*pfls*), i.e. a memory operation was accessing an address not present in physical memory. In this case, *cvmdispatch* invokes the page fault handler with parameters *cup* and $pcb[cup].edata$. For details on the paging algorithm used in Verisoft see [2].

5.4 CVM Primitives

The implementation of the various primitives is straightforward. Some of them are only updating the process control blocks of tasks and are therefore implemented in pure *C0*. Other primitives—e.g. those copying memory from one process to another—are manipulating data structures not visible in *C0*. In these cases, hardware-specific assembler code portions are inevitable. We inline them directly into the *C0* code with a special *ASM* statement.

5.5 Abstract Linker and Concrete Kernel

As we have seen in the sections before, it takes several things to build a concrete kernel from an abstract one. We have to provide implementations for these functions, the abstract kernel only declares (i.e. the CVM primitives), and we have to add functions that are not visible in the abstract kernel (i.e. *cvmdispatch*). Additionally, we have to add some extra global variables not needed in the abstract kernel.

Starting with two programs $A = (te_A, pt_A, gst_A)$ and $B = (te_B, pt_B, gst_B)$, we build the linked program $link(A, B) = (te_{ld}, pt_{ld}, gst_{ld})$ as follows: (i) We merge the two type name environments by simply adding one to the other. (ii) For any external procedure p in pt_A, i.e. with an empty body, we look for a corresponding procedure in pt_B with implementation and remove p from pt_A if one exists. Vice versa, we repeat this for procedures q in pt_B. pt_{ld} is then given by the disjoint union of the procedure tables updated as described afore. (iii) We build the new global symbol table by appending gst_B to gst_A: $gst_{ld} = gst_A; gst_B$. (iv) Finally, we scan all procedure bodies of pt_{ld} for external function calls denoted by the *C0* statement *ESCall*. For any of these statements we check, if it is now implemented after linking. If so, we replace the *ESCall* statement by a *SCall* statement. Linking does obviously not work for two arbitrary programs. Due to space limitations, we have omitted any preconditions here, e.g. the two symbol tables having to be disjoint.

We can now build a compilable concrete kernel by linking the CVM implementation *cvm* to the abstract kernel implementation *ak*: $ck = link(cvm, ak)$.

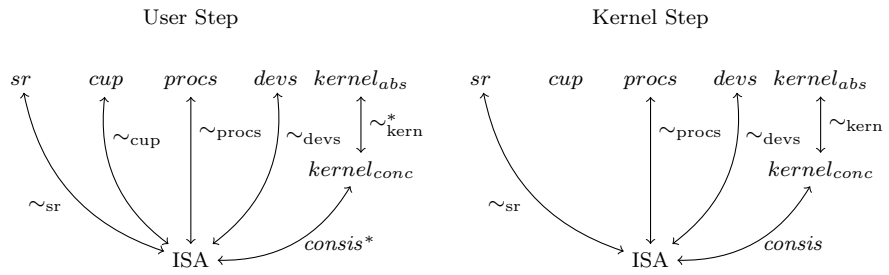

Fig. 4. Simulation Relations During User and Kernel Execution

6 CVM Implementation Correctness

6.1 User Process Relations

The implementation correctness of the CVM specification user process part $c_{\text{CVM}}.up$ is defined by three separate relations. $\sim_{cup}$ relates the current user process $c_{\text{CVM}}.cup$ to the value stored at the appropriate address in c_{phys}. $\sim_{sr}$ relates the status register $c_{\text{CVM}}.sr$ to the value stored at the appropriate address in c_{phys}. Last but not least, $\sim_{procs}$ defines the way, the user process configurations $c_{\text{CVM}}.procs(i)$ are to be stored in c_{phys}.

A physical machine with appropriate page fault handlers can simulate virtual machines. In Verisoft, we consider a simple pager that stores virtual memory in the swap memory, whereas the physical memory acts as a write back cache. The swap memory is provided by a designated hard disk with device ID $did_{shd} = 1$. For simplicity, we omit here the full hard disk model and consider only its content, a mapping of addresses to content: $sm : \mathbb{N} \rightharpoonup \mathbb{Z}$. Besides the architecturally defined physical memory address $pma(c_{\text{phys}}, va)$, we define a (software) swap memory address function $sma(c_{\text{phys}}, va)$ maintained by the page fault handler, which maps virtual addresses to addresses in $c_{\text{devs}}(1).sm$.

Let $?valid(c_{\text{phys}}, p, va)$ be a predicate denoting if a virtual address va for a process p lies in physical memory or not. Then we define a function *get_mm*, constructing the virtual memory of a process p from a physical machine configuration c_{phys} as follows: (i) for $?valid(c_{\text{phys}}, p, ad)$, we set $get_mm(c_{\text{phys}}, p)(ad) = c_{\text{phys}}.mm(pma(c_{\text{phys}}, ad))$ and (ii) $(c_{\text{devs}}(1)).sm(sma(c_{\text{phys}}, ad))$ else. Furthermore, we define a function *get_gpr* constructing the general purpose register file for a process p and a configuration c_{phys}: (i) if $cup = p \wedge c_{\text{phys}}.mode = 1$, we set $get_gpr(c_{\text{phys}}, p)(reg) = c_{\text{phys}}.gpr(reg)$, and (ii) $pcp[p].reg$, else, for $reg \in \mathbb{N}_{32}$. Correspondingly, we define functions *get_dpc*, *get_pcp*, *get_spr*, and the function $get_vm(c_{\text{phys}}, p)$ that combines the functions afore and returns a whole configuration.

The physical machine simulates a user process $i \in PID$, iff $get_vm(c_{\text{phys}}, i) = c_{\text{CVM}}.procs(i)$. We define $\sim_{procs}$ as the conjunction of this equality relation over all processes: $\sim_{procs} (c_{\text{CVM}}.procs, c_{\text{phys}}) = \bigwedge_{i=1}^{pmax} get_vm(c_{\text{phys}}, i) = c_{\text{CVM}}.procs(i)$.

6.2 Kernel Relations

First, the abstract kernel has to be simulated by the concrete kernel. Second, the concrete kernel is a *C0* program and cannot be executed directly on the hardware.

Thus, we depend on compiler correctness, i.e. a simulation relation between *C0* machines and physical machines.

6.2.1 Abstract Kernel and Concrete Kernel

We define a simulation relation $\sim_{kern}$ that tells us when a concrete kernel configuration *cc* encodes an abstract kernel configuration *ca*. As seen in Sect. 5, the concrete kernel has more variables and more function calls than the abstract one. Thus we define a mapping of abstract kernel variables $gvar^{ca}$ to concrete kernel variables $gvar^{cc}$ as $kalloc(g, hpm) :=$ (i) $gvar^{cc}_{gm}(v)$ for $g = gvar^{ca}_{gm}(v)$, (ii) $gvar^{cc}_{lm}(i + j, v)$ for $g = gvar^{ca}_{lm}(i, x)$, and (iii) $gvar^{cc}_{hm}(hpm(i))$ if $g = gvar^{ca}_{hm}(i)$. Note, that the constant j denotes the number of extra function calls in the concrete kernel and $hpm : \mathbb{N} \rightharpoonup \mathbb{N}$ is a mapping of heap indices in the abstract kernel to heap indices in the concrete kernel.

Now, we set $\sim_{kern}$ $(ca, cc, te_{ca}, pt_{ca}, te_{cc}, pt_{cc}, kalloc)$ iff (i) corresponding variables g^{ca} and $g^{cc} = kalloc(g^{ca}, hpm)$ have the same values and types, (ii) the recursion depths are equal modulo the constant number j of extra function calls in *cc*, (iii) the program rest of the abstract kernel *ca.pr* is a prefix of the concrete kernel program rest *cc.pr*, (iv) the abstract type name environment te_{ca} is a subset of the concrete one te_{cc}, and (v) all procedures declared or defined in the abstract procedure table pt_{ca} are also defined in the concrete pt_{cc}.

During user execution, the local memory stacks of both the abstract kernel and the concrete kernel are empty, as is the program rest of both kernels. This means, that $\sim_{kern}$ would be unprovable. Thus, we define a weaker form $\sim^*_{kern}$, which omits any properties on local variables, the recursion depth, and the program rest.

6.2.2 Compiler Correctness

The concrete kernel is written in C0 with inline assembler portions, while on the actual hardware, the translated object-code is executed. Hence, we have to define in a formal way, what correct translation of C0 means. Since our work is part of the Verisoft project, we are using the Verisoft simple non-optimizing C0 Compiler and the consistency relation it provides. Nevertheless, approaches like translation validation [25] are also feasible and have been successfully applied in other projects [19].

Compiler correctness is defined by means of a simulation relation $consis(te, pt, c_{\mathrm{C0}}, alloc, c_{\mathrm{ASM}})$ between configurations c_{C0} of *C0* machines and configurations c_{phys} of physical machines, which run the compiled code. Additionally, *consis* is parameterized with an allocation function *alloc* (a mapping of g-vars to memory addresses), a type name environment *te*, and a procedure table *pt*. A complete formal definition of *consis* with a correctness proof for a simple, non-optimizing compiler, can be found in [15].

Essentially, *consis* divides into three sub-relations:

(i) $consis_{code}(te, pt, c_{\mathrm{C0}}, c_{\mathrm{phys}})$, code consistency, requires that the compiled code is stored at a well-defined address in the machine configuration,

(ii) $consis_{c}(te, pt, c_{\mathrm{C0}}, c_{\mathrm{phys}})$, control consistency, requires that the program counters of the physical machine point to the start address of the code which has

been generated for the head of the program rest, and

(iii) $consis_d(te, pt, c_{C0}, alloc, c_{\text{phys}})$, data consistency. Data consistency states that g-vars are correctly stored in the physical machine and that some auxiliary information about stack and heap are stored correctly.

Like with $\sim_{kern}$, *consis* is too strong during user execution, Since the values stored in the registers of the physical machine are those of the current user process, some sub-relations do not hold or have to be modified at least: (i) Control consistency is discarded, because the program counters are related to the current user process. Since we always enter the kernel at the same entry point (cf. Sect. 4.5), we do not even store the old values when leaving system mode. (ii) During kernel execution, the relation between the size of the heap in the *C0* configuration and the one in the physical machine is defined by a designated general purpose register. Throughout user execution, we use the kernel variable *kheap* instead. We denote this—weaker—simulation relation by $consis^*$.

6.3 Device Relation

Since we are using a generalized device framework as introduced in Sect. 4.1, the devices as seen by the CVM model are nearly the same as those on the physical machine level, only the hard disk used as swap device ($did_{shd} = 1$) is not visible in CVM. Hence, $\sim_{dev}$ is merely an equivalence relation between the states of the CVM devices and those of the physical machine devices: $\sim_{\text{devs}}$: $\bigwedge_2^{dmax} c_{\text{CVM}}.devs(i) = c_{\text{devs}}(i)$

6.4 Correctness Theorem and Proof

We introduce one single simulation relation $\sim_{\text{CVM}}$ $(c_{\text{CVM}}, c_{\text{phys}}, c_{\text{devs}})$, for which we demand, that (i) the relations $\sim_{\text{sr}}$ and $\sim_{\text{procs}}$—and in the case of user mode $\sim_{\text{cup}}$—hold, (ii) there exists a concrete kernel configuration *cc*, such that $\sim_{\text{kern}}$ (and $\sim^*_{\text{kern}}$ respectively) and *consis* (and $consis^*$ respectively) hold. Furthermore, we denote initial configurations, i.e. the configurations after a reset, by a superscript 0.

Definition 6.1 [CVM Correctness] For all initial configurations $c^0_{\text{phys}}, c^0_{\text{devs}}, c^0_{\text{CVM}}$, and input sequences $ins_{\text{p\&d}}$ and for all steps i, there exists a function f, such that $c^i_{\text{CVM}} = \delta^i_{\text{CVM}}(c^0_{\text{CVM}}, f(ins_{\text{p\&d}}))$, and steps t, such that $(c^t_{\text{phys}}, c^t_{\text{devs}}) = \delta_{\text{p\&d}}(c^0_{\text{phys}}, c^0_{\text{devs}}, ins_{\text{p\&d}})$ with $\sim_{\text{CVM}}$ $(c^i_{\text{CVM}}, c^t_{\text{phys}}, c^t_{\text{devs}})$.

This correctness statement is proven by induction. The induction base case ($i = 0$) is defined by a CVM configuration c^0_{CVM}, whereas the kernel is running ($cvm.cup = 0$) and its program rest starts with *kdispatch* with parameter $eca = 1$ (for reset).

7 Status of the Formal Verification

The induction base case is already completely proven in Isabelle. For the induction step, we distinguish between user steps, abstract kernel steps, primitive steps and context switching. We have already obtained essential results by the formal verification of a paging mechanism [2], which represents the main difficulty for user steps.

To prove $\sim_{\mathrm{CVM}}$ for user steps by integrating these results appears to be work for another one or two months. For abstract kernel steps, $\sim_{\mathrm{procs}}$ has been shown, and here also the rest of the proof work is straightforward. Proving CVM primitives to be correct is tedious work due to the inline assembler portions. Nevertheless, for three of them we already have formal proofs in Isabelle [27]. Context switching, i.e. saving a process state to its process control blocks and restoring it, is also fully formally proven. All together, the CVM specification and the associated proofs comprise currently about 50,000 lines in Isabelle.

8 Summary and Further Work

We have presented a formal model for CVM and have defined the meaning of implementation correctness in this context. The pervasive formal correctness proof of the CVM implementation—which has been completed to a large extent—yields a trustworthy framework down to the hardware. Microkernel verification can now focus on verifying an abstract microkernel in a high-level language, avoiding to deal with tedious low-level argumentation but still with the benefits of pervasive systems verification.

Future work includes the verification of further CVM primitives, especially those dealing with devices. In particular, accessing devices in block mode, i.e. reading and writing big chunks with one kernel call, yields major challenges like handling interrupts that might occur during such accesses.

The new Hypervisor project in Verisoft XT deals with even more open research problems. Here, a multi-threaded virtualization layer, the hypervisor, runs multi-threaded on a multi-processor architecture with a weak memory model and is compiled using a highly optimizing compiler. Due to the major differences in design and complexity of this task, it seems unrealistic to expect anything of CVM to be reused but the experience and knowledge gained by the people involved in this work.

Yet, the applicability of our approach to smaller kernels has been shown with Verisoft's VAMOS. In the new Verisoft XT project, the commercial microkernel PikeOS [28] is to be verified on code level; unlike CVM, PikeOS might be interrupted in system mode, for instance a higher priority process might suspend a lower priority process' kernel call. This means that the CVM model has to be extended in order to deal with multiple kernel stacks. The success of this undertaking would prove, that the CVM approach is of considerable relevance for the huge market of embedded systems.

In order to achieve this, several obstacles have to be overcome from our experiences: (i) Code verification with an interactive theorem prover—though using a verification environment—is not applicable in a commercial setting due to the tremendous amounts of time it takes even for highly trained people. So far, automated tools are only useful for a restricted class of interesting properties. The degree of automation in software verification has to get close to that in hardware design. (ii) We are using a specially built and verified compiler in our work. Commercial, highly optimizing compilers are not verified and won't be for a couple of years. Different approaches of relating high-level code to object code like translation validation for optimizing compilers [19] or proof carrying code [18] are promising.

References

[1] Alkassar, E., M. Hillebrand, S. Knapp, R. Rusev and S. Tverdyshev, *Formal device and programming model for a serial interface*, in: B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, 2007, pp. 4–20.
URL http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-259/paper04.pdf

[2] Alkassar, E., N. Schirmer and A. Starostin, *Formal pervasive verification of a paging mechanism*, in: *14th International Conference, TACAS 2008, Proceedings (to appear)*, Lecture Notes in Computer Science (2008).

[3] Beuster, G., N. Henrich and M. Wagner, *Real world verification – Experiences from the Verisoft email client*, in: G. Sutcliffe, R. Schmidt and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006)*, CEUR Workshop Proceedings **192** (2006), pp. 112–125.

[4] Bevier, W. R., W. A. Hunt, Jr., J. S. Moore and W. D. Young, *An approach to systems verification*, Journal of Automated Reasoning **5** (1989), pp. 411–428.

[5] Beyer, S., "Putting It All Together: Formal Verification of the VAMP," Ph.D. thesis, Saarland University, Computer Science Department (2005).

[6] Beyer, S., C. Jacobi, D. Kroening, D. Leinenbach and W. Paul, *Putting it all together: Formal verification of the VAMP*, International Journal on Software Tools for Technology Transfer **8** (2006), pp. 411–430.

[7] Dalinger, I., "Formal Verification of a Processor with Memory Management Units," Ph.D. thesis, Saarland University, Computer Science Department (2006).

[8] Dalinger, I., M. Hillebrand and W. Paul, *On the verification of memory management mechanisms*, in: D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, Lecture Notes in Computer Science **3725** (2005), pp. 301–316.

[9] Heiser, G., K. Elphinstone, I. Kuz, G. Klein and S. Petters, *Towards trustworthy computing systems: taking microkernels to the next level*, Operating Systems Review (2007).

[10] Hillebrand, M., "Address Spaces and Virtual Memory: Specification, Implementation, and Correctness," Ph.D. thesis, Saarland University, Computer Science Department (2005).

[11] Hillebrand, M., T. In der Rieden and W. Paul, *Dealing with I/O devices in the context of pervasive system verification*, in: *ICCD '05* (2005), pp. 309–316.
URL http://www.iccd-conference.org/proceedings/2005/049_hillebrandm_dealing.pdf

[12] Hohmuth, M. and H. Tews, *The* VFiasco *approach for a verified operating system*, Technical Report TUD-FI05-15, Dresden University of Technology, Department of Computer Science (2005).

[13] In der Rieden, T. and S. Knapp, *An approach to the pervasive formal specification and verification of an automotive system*, in: *FMICS '05* (2005), pp. 115–124.

[14] Knapp, S. and W. Paul, *Pervasive verification of distributed real-time systems*, in: T. H. M. Broy, J. Grünbauer, editor, *Software System Reliability and Security*, IOS Press, NATO Security Through Science Series. Sub-Series D: Information and Communication Security **9**, 2007, pp. 239–297.

[15] Leinenbach, D. and E. Petrova, *Pervasive compiler verification – from verified programs to verified systems*, in: *3rd intl Workshop on Systems Software Verification (SSV08), to appear* (2008).

[16] Liedtke, J., *On micro-kernel construction*, in: *Proceedings of the 15th ACM Symposium on Operating systems principles (SOSP 1995)* (1995), pp. 237–250.

[17] Mueller, S. M. and W. J. Paul, "Computer Architecture: Complexity and Correctness," Springer, 2000.

[18] Necula, G. C., *Proof-carrying code*, in: *POPL*, 1997, pp. 106–119.

[19] Necula, G. C., *Translation validation for an optimizing compiler*, in: *PLDI*, 2000, pp. 83–94.

[20] Neumann, P. G. and R. J. Feiertag, *PSOS revisited*, in: *ACSAC* (2003), pp. 208–216.

[21] Ni, Z., D. Yu and Z. Shao, *Using xcap to certify realistic systems code: Machine context management*, in: K. Schneider and J. Brandt, editors, *TPHOLs*, Lecture Notes in Computer Science **4732** (2007), pp. 189–206.

[22] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science **2283**, Springer, 2002.

[23] Petrova, E., "Verification of the C0 Compiler Implementation on the Source Code Level," Ph.D. thesis, Saarland University, Computer Science Department (2007).

[24] Pfitzmann, B., J. Riordan, C. Stüble, M. Waidner and A. Weber, *The perseus system architecture*, in: D. Fox, M. Köhntopp and A. Pfitzmann, editors, *VIS 2001, Sicherheit in komplexen IT-Infrastrukturen* (2001), pp. 1–18.

[25] Pnueli, A., M. Siegel and E. Singerman, *Translation validation*, Lecture Notes in Computer Science **1384** (1998), pp. 151+.
URL citeseer.ist.psu.edu/article/pnueli98translation.html

[26] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technical University of Munich (2006).

[27] Starostin, A. and A. Tsyban, *Correct microkernel primitives* (2008).

[28] SYSGO AG, *PikeOS - embedded system software for safety critical real-time systems, rtos and embedded linux*, http://lists.sysgo.com/en/products/pikeos/ (2007).

[29] The Verisoft Consortium, *The Verisoft Project*, http://www.verisoft.de/ (2003).

[30] Tuch, H. and G. Klein, *Verifying the L4 virtual memory subsystem*, in: G. Klein, editor, *Proceedings of the NICTA Formal Methods Workshop on Operating Systems Verification* (2004), pp. 73–97.

[31] Tuch, H., G. Klein and M. Norrish, *Types, bytes, and separation logic*, in: M. Hofmann and M. Felleisen, editors, *POPL* (2007), pp. 97–108.

[32] Walker, B. J., R. A. Kemmerer and G. J. Popek, *Specification and verification of the UCLA unix security kernel*, Commun. ACM **23** (1980), pp. 118–131.

Correct Microkernel Primitives

Artem Starostin[a,1] Alexandra Tsyban[a,2]

[a] *Computer Science Department, Saarland University*
P.O. Box 15 11 50, 66041 Saarbrücken, Germany

Abstract

Primitives are basic means provided by a microkernel to implementors of operating system services. Intensively used within every OS and commonly implemented in a mixture of high-level and assembly programming languages, primitives are meaningful and challenging candidates for formal verification. We report on the accomplished correctness proof of academic microkernel primitives. We describe how a novel approach to verification of programs written in C with inline assembler is successfully applied to a piece of realistic system software. Necessary and sufficient criteria covering functional correctness and requirements for the integration into a formal model of memory virtualization are determined and formally proven. The presented results are important milestones on the way to a pervasively verified operating system.

Keywords: Formal Verification, Theorem Proving, Operating System, Microkernel, Inline Assembler.

1 Introduction

Correctness guarantees for computer systems is a hot research topic. Since there are a lot of examples when the correctness of separate computer components has been successfully established, the formal verification of an entire industrial-size system is being brought to the forefront. In [8] Moore, the head of the famous CLI project, proposes the grand challenge of whole computer system pervasive verification.

Verisoft [3] is a research project inspired by the problem of a complete computer system correctness. The project aims at the development of the pervasive verification technology [10] and demonstrating it by applying to an exemplary computer system. A prototypic system comprises (i) a pipelined microprocessor with memory management units, (ii) a number of devices, in particular, a hard disk, (iii) a microkernel, (iv) a simple operating system, and (v) an exemplary user application. Pervasive formal verification of the whole system is attempted. The process is supported by a variety of computer aided verification tools, both interactive and

[1] Email: starostin@wjpserver.cs.uni-sb.de. Work was supported by the International Max Planck Research School for Computer Science (IMPRS).

[2] Email: azul@wjpserver.cs.uni-sb.de. Work was supported by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project.

[3] www.verisoft.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

automated, in order to minimize the possibility of errors induced by verification engineers.

This work relates to the problem of operating system microkernel correctness. A microkernel is the minimal kernel which, basically, provides no operating system services at all, but only the mechanisms necessary to implement such services. The mechanisms include process and memory management, address spaces, low-level IPC, and I/O. Usually, they are implemented in the form of *primitives*, microkernel routines which provide this functionality to the upper layers. Since every service of an operating system makes use of primitives, the correctness of the latter becomes of special importance.

In the current paper we discuss the correctness issues of primitives of an academic operating system microkernel. We describe how the methodology for the system software verification developed in the frame of Verisoft is successfully applied to primitives implemented in C with inline assembler. We outline the correctness criteria of microkernel primitives. Stating the correctness theorems we show what it means that a primitive fulfills these correctness criteria. We sketch a general idea how such theorems are proven. In a case study we elaborate on particular for the example details of specifications and proofs.

The contribution of this paper is that (i) *all* necessary and sufficient correctness criteria of primitives of a microkernel for a pervasively verified system are determined and formally proven, (ii) a novel, convenient for formal use, approach to verification of C with inline assembler programs is presented, and (iii) an important part of a realistic microkernel is proven correct showing that seamless formal verification of crucial parts of operating systems is feasible. All material presented in the paper is supported by formal theories in a computer aided theorem prover.

Related Work

A number of research projects suggest ideas to microkernel verification. Choosing reasoning either in C or assembler semantics, to the best of our knowledge, nobody exploits their combination. The L4.verified project, targets at constructing seL4 [4], a formally verified operating system kernel. From the system's prototype designed in Haskel both formal model and C implementation are generated. A richer compared to Verisoft subset of C including pointer arithmetic is used, which, however, provides less expressive semantics than inline assembler as the latter makes possible to accesses even registers of a processor. A substantial progress seems to be achieved in the verification of the model, but only exemplary parts of the source code are reported verified. The FLINT project exploits an x86 assembly code verification environment for certification of context switching routines [9], an important microkernel part. No results on integration of object code correctness into a high-level programming language are reported. The recent Robin project aims at the verification of Nova microhypervisor [12]. Although implementation is in (a subset of) C++ with inline assembler, the verification is planned to cover only C++ parts. Currently there is no connection to real object code, which seems to be planned for the (far) future. It is planned to build a model precise enough to catch virtual memory aliasing and address space separation errors, however it is unclear whether these properties will be shown to be respected by the hypervisor's implementation.

Outline

In Sect. 2 we discuss implementation issues and formal model of a microkernel. We briefly formalize all concepts necessary to present the microkernel correctness criteria which have to be satisfied by its primitives. Next, in Sect. 3, we elaborate on our verification methodology and sketch the semantics of C programs with inline assembler parts. In Sect. 4 we proceed with the correctness theorem for a primitive. The presented approach is supported by the case study in Sect. 5 for which the primitive that copies data between processes is selected. We conclude in Sect. 6.

Notation

We denote the set of boolean values by $\mathbb{B}$ and the set of natural numbers including zero by $\mathbb{N}$. We denote the set of natural numbers less then x by $\mathbb{N}_x$. We denote the list of n elements of type T by T^n. The elements of a list x are accessed by $x[i]$, its length is denoted by $|x|$. The operator $\langle x \rangle$ yields for a bit string $x \in \mathbb{B}^n$ the natural number represented by x. We allow to interchange a bitvector x with its value $\langle x \rangle$. The set of all possible configurations of a concept x is defined by C_x.

2 An Academic Operating System Microkernel

We consider an exemplary academic microkernel which provides mechanisms for the (i) process and memory management, (ii) address spaces, (iii) IPC, and (iv) device communication.

2.1 Implementation Issues

The microkernel implements the *Communicating Virtual Machines (CVM)* [3] model which defines the parallel execution of concurrent user processes interacting with a kernel. According to the model the microkernel is split into two logical parts: (i) the *abstract* kernel which provides an interface to a user or an operating system and could be implemented in a pure high-level programming language, and (ii) the *lower layers* which implement the desired functionality stated in the beginning of Sect. 2. The implementation of the low-level functionality necessarily contains assembler portions because processor registers and user processes could not be accessed by ordinary C variables. By linking the two kernel parts together the *concrete* kernel, a program which can run on a target machine, is obtained.

The kernel lower layers could be split into three logical parts: (i) primitives, (ii) a page fault handler, and (iii) context switch routines. Within the paper we discuss the correctness of primitives. They are implemented in the C0 programming language [7], a slightly restricted C, with inline assembler parts. In brief, the limitations of C0 compared to standard C are as follows. Prefix and postfix arithmetic expressions, e.g., $i{+}{+}$, are forbidden, as well as function calls as a part of expressions. Pointers are typed and do not point to local variables or to functions. Void pointers and pointer arithmetic are not supported. The size of arrays has to be statically defined.

Name	Description	Comment
copy	copies data between processes	A
phys_copy	copies data between virtual and the physical memory	A
get_vm_word	reads a word from the virtual memory of a process	A
set_vm_word	writes a word to the virtual memory of a process	A
out_word	writes a word to a device	AD
in_word	reads a word from a device	AD
virt_io	copies data between a device and a process	AD
phys_io	copies data between a device and the physical memory	AD
phys_io_range	I/O operations on port ranges	AD
reset	initializes the memory and the registers of a process	
get_vm_gpr	reads a register of a process	
set_vm_gpr	writes a register of a process	
alloc	gives additional memory to a process	
free	releases a given amount of the memory of a process	
clone	clones a process	
set_mask	mask external interrupts	

Table 1
List of primitives of the microkernel

2.2 Primitives

The academic microkernel contains 16 primitives described in Table 1. The comment 'A' denotes that a primitive has an inline assembler portion. The comment 'D' designates that a primitive accesses devices. Thus, the primitives can be divided into three groups: (i) 7 primitives implemented in pure C0, (ii) 4 primitives which have assembler portions, and (iii) 5 primitives which have assembler portions and access devices. In this paper we give the methodology for verification of code written in C0 with inline assembler. It is applicable to all the primitives. However, we have verified so far primitives from the second group.

2.3 A Formal Model

The CVM model defines a parallel execution of the kernel and N user processes on an underlying physical machine with a hard disk. According to CVM, the C0 language semantics is used to model the computation of the kernel, and semantics of virtual machines models the computation of user processes. In the following, we outline the necessary concepts of the model: (i) physical and virtual machines [3], (ii) a hard disk [5], and (iii) C0 machines [7]. Having them, we sketch the CVM semantics and give its correctness criteria. For details cf. [6]. Memories of physical and virtual machines are conceptually organized in pages of P machine words.

2.3.1 Physical Machines

Physical machines are the sequential programming model of the VAMP hardware [2] as seen by a system software programmer. They are parameterized by (i) the set $SAP \subseteq \mathbb{B}^5$ of *s*pecial purpose register *a*ddresses visible to *p*hysical machines, and (ii) the number TPP of *t*otal *p*hysical memory *p*ages which defines the set $PMA = \{a \mid 0 \leq \langle a \rangle < TPP \cdot P\} \subseteq \mathbb{B}^{30}$ of accessible *p*hysical *m*emory *a*ddresses. The machines are 5-tuples $pm = (pc, dpc, gpr, spr, m)$ with the following components: (i) the normal $pm.pc \in \mathbb{B}^{32}$ and the delayed $pm.dpc \in \mathbb{B}^{32}$ program counters used to implement the delayed branch mechanism, (ii) the general purpose $pm.gpr \in \mathbb{B}^5 \mapsto \mathbb{B}^{32}$ and the special purpose $pm.spr \in SAP \mapsto \mathbb{B}^{32}$ register files, and (iii) the word addressable physical memory $pm.m \in PMA \mapsto \mathbb{B}^{32}$.

The computation is possible in two modes: user and system. In user mode a memory access to a virtual address va is subject to address translation. It either redirects to the translated 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 $v(pm, va)$ maintained by the memory management unit of the physical machine. When on, it signals that the page storing the virtual address va resides in the main memory, elsewise, it is on a hard disk.

The semantics of an uninterrupted execution is defined by the underlying *i*nstruction *s*et *a*rchitecture (ISA). On an interrupt signal, which could be internal or external, the machines switches to the system mode and invokes a special piece of software—an interrupt handler. Within the paper, we are interested in two particular kinds of interrupts: (i) page faults, and (ii) system call exceptions. A page fault is treated by the *page fault handler*, a routine which translates addresses and loads missing pages from a hard disk into the physical memory. Its implementation servers several purposes. For instance, it could be used to handle a page fault and to guarantee that no page fault will occur within a certain period in the future. The latter property is needed for the primitives, thus, they heavily call the handler (for details cf. Sect. 2.5). System call exceptions occur due to a special instruction, called the *trap*. It is used by an assembler programmer in order to invoke one of the system calls provided by the operating system microkernel. System calls, viewed from a simplified perspective, are just the wrappers around the primitives.

2.3.2 Virtual Machines

Virtual machines are the hardware model visible for user processes. They give user an illusion of an address space exceeding the physical memory. No address translation is required, hence page faults are invisible. The virtual machine's parameters are: (i) the number TVP of *t*otal *v*irtual memory *p*ages which defines the set of accessible *v*irtual *m*emory word *a*ddresses $VMA = \{a \mid 0 \leq \langle a \rangle < TVP \cdot P\} \subseteq \mathbb{B}^{30}$, and (ii) the set $SAV \subseteq SAP$ of *s*pecial purpose registers *a*ddresses visible to *v*irtual machines. Their configuration, formally, is a 5-tuple $vm = (pc, dpc, gpr, spr, m)$ where only $vm.spr \in SAV \mapsto \mathbb{B}^{32}$ and $vm.m \in VMA \mapsto \mathbb{B}^{32}$ differ from the physical machines. Semantics is completely specified by the ISA with the following exception. Due to safety reasons we split the set SAV into two parts: (i) the set SAV_R of *read only* register addresses, and (ii) the set SAV_W of addresses of registers that could be *completely* accessed by a user. A write attempt to a register $vm.spr[x]$

with $x \in SAV_R$ has no effect. The set SAV_R contains the register *ptl* (page table length). It stores the amount of virtual memory allocated to the process measured in pages. We abbreviate $vm.spr[ptl] = vm.ptl$.

2.3.3 Integrating a Hard Disk

We use the formal model of a hard disk based on the ATA/ATAPI protocol. We denote the configuration of the hard disk by *hd*. Only the component $hd.sm \in \mathbb{N}_{2^{30}} \mapsto \mathbb{N}_{2^{32}}$ which models the disk content as a word-addressable memory is of our interest. A step of the system (pm, hd) comprising the physical machine and the hard disk is denoted by the function $\delta(pm, hd) = (pm', hd')$. If no write instruction to the disk is executed only the physical machine is updated according to its semantics. Otherwise, both *pm* and *hd* are changed.

2.3.4 C0 Machines

A C0 machine is a 6 tuple $c = (pr, tt, ft, rd, lms, hm)$. Its components are: (i) the program rest $c.pr$, a sequence of statements which still has to be executed, (ii) the typetable $c.tt$ which collects information about types used in the program, (iii) the function table $c.ft$ storing information about functions of the program, (iv) the recursion depth $c.rd$, (v) the local memory stack $c.lms$ mapping numbers $i \leq c.rd$ to memory frames which implement a relatively low-level memory model and comprise components for the number of variables in a frame, their names, types, and contents, and (vi) the heap memory $c.hm$ which is a memory frame as well.

The global memory of a C0 machine c is $c.lms(0)$. The top local memory frame is denoted by $top(c) = c.lms(c.rd)$. A memory frame first includes the parameters of the corresponding function. A variable of a machine c is a pair (m, i), where m is a memory frame and i is the number of the variable in the frame. By $va(c, i) = (top(c), i)$ we denote the i-th variable of the current function context.

2.3.5 Communicating Virtual Machines

The CVM configuration is formally a triple $cvm = (up, ak, cp)$ with the following components: (i) the list of N user processes $cvm.up \in C_{vm}^N$ represented by virtual machines, (ii) the abstract kernel $cvm.ak \in C_c$ modeled by a C0 machine, and (iii) the current process identifier $cvm.cp \in \mathbb{N}_N$, where $cvm.cp = 0$ stands for the kernel. The CVM semantics distinguishes user and kernel computations. In case $cvm.cp \neq 0$ the user process $pid = cvm.cp$ is intended to make a step. In case no interrupt occurs it boils down to the step of the virtual machine $cvm.up[pid]$. Otherwise, the kernel dispatcher is invoked and the kernel computation starts. The kernel dispatcher handles possible page faults and determines whether a primitive f is meant to be executed. In case it is, the parameters of the primitive p_f are extracted by means of the system call mechanism. The specification f_S is applied to the user processes $cvm'.up = f_S(cvm.up, p_f)$. Next, the user computation resumes.

2.4 Correctness Criteria

Microkernel correctness requirements have to relate: (i) the implementation of kernel lower layers, encoded by the C0 machine c, (ii) the CVM model *cvm*, and (iii) the

physical machine with the hard disk (pm, hd).

The implementation c is related to the CVM model by means of linking. We use the formal specification of the linking operator $link(cvm.ak, c) = k$. It takes two C0 machines encoding the abstract kernel and the implementation of its lower layers, respectively, and produces the concrete kernel k, also a C0 machine. We state that the concrete kernel k correctly runs on the physical machine pm by means of the C0 compiler consistency relation (cf. Sect. 3.2).

The correctness criteria for the user processes is hidden inside the memory virtualization relation. This simulation relation, called the $\mathcal{B}$-relation, specifies a parallel execution of the user processes *cvm.up* on one physical machine *pm*. In order to specify the $\mathcal{B}$-relation, let us first give a notion of a *process control block (PCB)*. The PCBs are C0 data structures permanently residing in the memory of the underlying physical machine. They store the information about visible registers of all user processes. Thus, we are able to reconstruct user virtual machines from the contexts stored in the PCBs. The function $virt(pid, pm, hd) = vm$ yields the virtual machine for the process *pid* by taking the register values from the corresponding PCB fields. The memory component *vm.m* of the built virtual machine is constructed out of the physical memory and the data on the hard disk depending on where a certain memory page lies. Formally, $vm.m[a] = pm.m[pma(pid, a)]$ if $v(pm, a)$, otherwise $vm.m[a] = hd.sm[sma(a, pid)]$. The physical memory address is computed by the function $pma(pid, a)$ while the swap memory address is yielded by the function $sma(a, pid)$ (for the definitions cf. [1,6]). Then, the $\mathcal{B}$-relation is defined formally as follows: $\mathcal{B}(cvm.up, pm, vm) = \forall pid \in \mathbb{N}_N \ : \ virt(pid, pm, hd) = cvm.up[pid]$. There is a number of additional correctness demands omitted due to the space limitations.

2.5 A Page Fault Handler

The $\mathcal{B}$-relation can only be maintained with an appropriate page fault handler. The page fault handler is a routine which serves two purposes. Called for a virtual address *va* and a process identifier *pid* it (i) yields to the caller the translated physical memory address $pma(pid, va)$, and (ii) guarantees that the page storing $pma(pid, va)$ resides in the physical memory of the machine running the handler.

Possibly called twice in a primitive in order to translate addresses for different processes, it must respect the following. An appropriate page fault handler must not swap out the memory page that was swapped in during a previous call to it. In order to guarantee this a proper page eviction strategy must be used. We support two lists, called *active* and *free*, for the page management. Together they describe all pages of physical memory accessible to a user. Items of the free list describe the pages that immediately could be given to user, i.e., without swapping out a page to the hard disk. Active list describes physical pages that store a virtual page. When all physical memory is occupied, a page from the active list is evicted and replaced by the one loaded from the hard disk according to the FIFO strategy. For formal details and correctness issues cf. [1].

3 Verification Approach

There are several possibilities to argue about the correctness of kernel lower layers, and in particular of primitives. One might have an idea to reason about their object code in the machine language semantics. Due to the huge size of the target code—the kernel lower layers translated by the C0 compiler are 11K lines long—this approach seems to be unfeasible for the interactive verification. Running to extremes, one can try to verify system code on a very high-level of abstraction, e.g., by means of a generic verification environment for imperative programs [11], and then transfer the results down to the necessary level introducing refinement theorems. However, techniques that allow reasoning about mixture of C and assembler code in such environments were only recently invented (the approach is used in [1]). They basically aim at big C programs with assembler portions, isolated in separate functions. Since this is not the case for the primitives—they are relatively small C functions which can have several inline assembler parts—we decided to do the formal verification in a mixture of C0 small step and inline assembler semantics.

3.1 Verification Environment

We use Isabelle/HOL theorem prover as the basis for the verification environment. All the concepts and their semantics listed in Sect. 2.3 were formalized in Isabelle by the colleagues in the Verisoft project. The source code of the primitives is automatically translated by a tool into the C0 small step semantics in Isabelle.

3.2 C with Inline Assembler Semantics

A C0 configuration c is related to the underlying physical machine pm by the compiler simulation relation $consis(alloc)(c, pm)$ parameterized over an allocation function $alloc$ which maps C0 variables to the physical memory cells. Essentially, the relation is a conjunction of the following facts: (i) *value* consistency: the respective variables of c and pm have the same values and the reachable portions of the heaps in $c.hm$ and $pm.m$ are isomorphic, (ii) *control* consistency: the delayed program counter $pm.dpc$ points to the start of the translated code of the first statement of $c.pr$ and $pm.pc = pm.dpc + 4$, (iii) *code* consistency: the compiled code lies at the correct address in the memory $pm.m$, and (iv) *stack* consistency: the heap resp. stack pointers which reside in the registers $pm.gpr[29]$ resp. $pm.gpr[30]$ point to the first free address of $c.hm$ resp. to the beginning of $top(c)$. For details cf. [7].

An assembler instruction list il can be integrated by a special statement $asm(il)$ into the C0 code. As long as no such statement occurs the C0 semantics is applied. The former approach to deal with verification of an assembler statement is to maintain the compiler consistency relation with execution of every single instruction from il (cf. Sect. 4.3 of [3]). This method turned out to be inconvenient due to excessive complexity of formal proofs, therefore a new one was developed and used.

In brief, the novel approach is as follows. On an assembler statement the execution is switched to the consistent underlying physical machine and continues directly there. When the assembler instructions have been executed we switch back to the C0 level. For this we have to update the C0 machine possibly affected by the

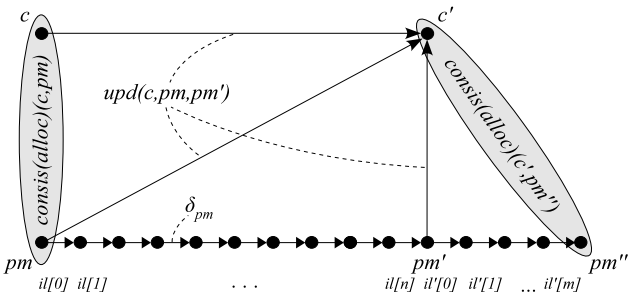

Figure 1. Switching C and assembler semantics

assembler instructions. The allocation function *alloc* makes it possible to determine which variables of the C0 machine 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 w.r.t. the allocation function *alloc*, i.e., $consis(alloc)(c, pm)$. From the consistency relation we have that the program counters of pm point to the address of the assembler statement: $pm.dpc = ad(asm(il))$ and $pm.pc = pm.dpc + 4$, where $ad(s)$ yields for a statement s its address in the memory of pm. This allows us to start reasoning about the correctness of the assembler 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 machine 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' (cf. Fig. 1). A number of restrictions are imposed on the changes in the physical machine, which guarantee that the C0 machine is not destroyed by the assembler 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 memory region where the compiled code is stored stays the same, i.e., we forbid self-modifying code, (iii) the stack and heap pointers are unchanged: $pm'.gpr[x] = pm.gpr[x]$ for $x \in \{29, 30\}$, (iv) the memory occupied be the local memory frames remains the same except for $top(c)$, and (v) pointers change is forbidden except setting them to *null*. We formally prove that we deal with assembler portions which meet these restrictions.

The program rest is updated straightforwardly—the assembler statement is removed, i.e., $c'.pr = r$. The memory update proceeds separately for the global, the top local, and the heap memories. For each of them the respective memory cells of the physical machines configurations pm and pm' are compared. In case a memory cell at an address a is changed, the value of the variable x, s.t. $alloc(c', x) = a$ is updated with $pm'.m[a]$. However, the compiler correctness relation does not necessarily hold between the C0 configuration c' and the physical machine pm'. The control consistency will be broken if the assembler statement $asm(il)$ is either (i) the last statement of a loop body, or (ii) the last statement of the 'then' part of a conditional statement. The translation of these statements to the target code results in a list of assembler instructions il' which has to be executed by the machine pm' in order to reach a consistent to c' state. Note that il' contains only control instructions, and, hence does not affect any C0 variable. Executing il' we transit from pm' to pm'' updating the program counters and regain consistency $consis(alloc)(c', pm'')$.

4 Correctness of a Primitive

Since primitives are parts of the microkernel, their correctness is closely related to the correctness of the whole kernel. Execution of a primitive is one of the induction cases of the overall kernel correctness theorem [6]. We distinguish two main theorems for each primitive: (i) the primitives functional correctness, and (ii) the top-level correctness of a primitive. The latter is used to prove the induction case of the overall kernel correctness theorem and, therefore, claims correctness criteria needed for the integration, for instance that the abstract kernel data is not corrupted. The former is used as an auxiliary theorem to prove the latter. It states the correctness of the input-output relation of a primitive call. Such modularization increases the robustness of formal theories to the possible code changes, e.g., due to the errors disclosed during the verification. In this case, one has to adapt the proofs only of the first theorem which is much simpler than the second one. Next, we present the general idea behind these theorems and discuss their formal proofs.

4.1 Functional Correctness

The functional correctness justifies the input/output relation of a primitive. We start in some C0 state k encoding the concrete kernel and consistent to the underlying physical machine pm and claim the requirements $pre_f(k, pm)$ to a primitive f caller. We end in the resulting state obtaining the desired values $post_f(k', pm')$ of C0 variables and memory cells of the physical machine. Note that both pre- and postconditions, in general, speak not only about values of C0 variables, but also about the memory parts of the underlying machine which are not accessible via variables but are subject to change by inline assembler code. The straightforward idea of the functional correctness is reflected in the next theorem.

Theorem 4.1 (Functional Correctness of a Primitive) Let k be the concrete kernel calling the primitive f with the parameters p_f: $k.pr = f(p_f); r$. Let (pm,hd) be the configuration of the underlying physical machine with the hard disk, s.t. it is consistent to the concrete kernel: $consis(alloc)(k, pm)$. Assume that the precondition $pre_f(k, pm)$ to the primitive is satisfied, then there exist (i) a number of steps T of the physical machine with the hard disk, s.t. $(pm', hd') = \delta^T(pm, hd)$, and (ii) a configuration of the concrete kernel k' with an appropriate allocation function $alloc'$, s.t. they are consistent to the physical machine: $consis(alloc')(k', pm')$, and the desired postcondition $post_f(k', pm')$ holds.

In our experience it is inconvenient to prove such theorems directly. We rather create several separate lemmas of the same form but speaking about the code in *different* semantics. For example, if a primitive contains a number of assembler instructions wrapped both from the beginning and the end in C0 code, we create three lemmas: one for the C0 part before assembler, next for the assembler portion, and, finally, for the remaining C0 part. This simple idea is easily scalable to arbitrary combinations of C and assembler. We prove such lemmas by applying C0 and inline assembler semantics. The crucial point is the construction of a consistent C0 machine after the assembler part execution. We proceed as described in Sect. 3.2.

The verification proceeds, certainly, with respect to the total correctness criteria,

Figure 2. Memory structure of the microkernel claimed by the kernel invariant

i.e., we show the termination and the absence of run-time errors. The machinery for this is hidden inside the C0 small step semantics. The set C_c of all possible C0 configurations is represented formally in Isabelle by the *option* type which extends C_c with an additional *error* state. The semantics is constructed in a way that the computation ends in a non-error state only in the case that no run-time errors occur. We formally show that the resulting configuration of a primitive call is not in the faulting state. We do this in an iterative fashion, i.e., we show that the execution of every single statement brings some sensible configuration. This could happen only in case all expressions of the statements are correctly evaluated. The correctness demands to the expression evaluation forces us to show formally that neither null-pointer dereference nor out-of-boundary array access happens. This also proves the termination of single statements. In order to guarantee the termination of a whole program provided that statements terminate we have to show that neither infinite loops nor infinite recursive calls occur. Since we do not use recursion in the kernel implementation, we pay attention only to loops. Their termination is closely related to the way loops are verified in the C0 semantics. The correctness of a loop is established by an inductive lemma. We formally specify the number of the loop iterations by a ranking function over the variables modified in the loop. Since we proceed by induction on the result yielded by the ranking function, the termination follows. We give details in the example (cf. Sect. 5.3). The absence of run-time errors in assembler portions boils down to the absence of interrupts conditions which are required to the proven by the inline assembler semantics. The termination of assembler loops is proven analogously to C0 loops.

4.2 Correctness in the Context of the Kernel

The correctness criteria needed for the integration are, basically, split into two parts. They are: (i) the kernel correctness requirements stated in Sect. 2.4, and (ii) the *kernel invariant* which turns out to be necessary to be proven first. The kernel invariant $inv(cvm, k, pm, hd)$ is the conjunction of the following statements: (i) the memory map properties, (ii) the page fault handler invariants, (iii) the validity of the C0 machine encoding the concrete kernel, and (iv) the hard disk properties and liveness requirements for the system 'hard disk - physical machine'.

4.2.1 Memory Map Properties

The kernel code has a particular alignment in the memory. Its data structures lie both in the global and the heap memories. For safety reasons we must know these regions, and know which of their parts could be changed with every step of the kernel, for instance with the execution of a primitive. Fig. 2 depicts the memory structure which we describe formally.

4.2.2 Page Fault Handler Invariants

As mentioned in Sect. 2.5, the page fault handler is (heavily) called by the kernel. The handler maintains a variety of global data structures, in particular lists for page management. Therefore, we must claim that no functions besides the page fault handler are allowed to modify its data structures. Due to the complexity of the page fault handler its verification is attempted by means of the refinement technique which connects its representation on several semantical layers. In order to support that approach, we formally preserve: (i) the mapping between the implementation of the kernel lower layers c which contains the handler and the PFH abstraction, and (ii) the validity properties over the handler abstraction. The page fault handler properties relevant for the primitives correctness comprise: (i) for distinct pairs $(pid_1, va_1) \neq (pid_2, va_2)$ the translated physical addresses are distinct: $pma(pid_1, va_1) \neq pma(pid_2, va_2)$, (ii) every physical address is associated exactly with one pair (pid, va), and (iii) all translated addresses lie outside the kernel range: $\forall(pid, va) : pma(pid, va) \notin [0 : KERNEL_END)$.

Next, we present the top-level correctness theorem of a primitive execution. It turns out that its proof requires several static properties $pro(cvm.ak, c)$ over the abstract kernel and the implementation of the kernel lower layers. They are the necessary preconditions to a correct linking and state, not exclusively, the following: (i) the function tables *cvm.ak.ft* and *c.ft* encode the same function signatures, (ii) all external function declarations in *cvm.ak.ft* have an implementation in *c.ft* and vice versa, and (iii) the type tables *cvm.ak.tt* and *c.tt* encode the same types.

Theorem 4.2 (Top-level Correctness of a Primitive) Let k be the concrete kernel calling the primitive f with the parameters p_f: $k.pr = f(p_f); r$. Let *cvm* be the configuration of the CVM model, and (*pm*,*hd*) be the configuration of the underlying physical machine with the hard disk. Assume that (i) the concrete kernel is consistent to the physical machine: $consis(alloc)(k, pm)$, (ii) the relation $\mathcal{B}(cvm.up, pm, hd)$ holds, (iii) the preconditions $pre_f(k, pm)$ to the primitive are satisfied, (iv) the kernel invariant $inv(cvm, k, pm, hd)$ holds, and (v) the kernel static properties $pro(cvm.ak, c)$ are satisfied, then there exists a number of steps T of the physical machine with the hard disk, s.t. $(pm', hd') = \delta^T(pm, hd)$ after which (i) the CVM model executes the primitive and the relation $\mathcal{B}(f_S(cvm.up, p_f), pm', hd')$ still holds, (ii) the concrete kernel executes the primitive and is still consistent to the physical machine: $\exists k', alloc' : consis(alloc')(k', pm') \wedge k'.pr = r$, and (iii) the kernel invariant is preserved: $inv(cvm', k', pm', hd')$.

5 Case Study: Copying Data Between Processes

As an application of the developed approach we show how we establish the correctness of the *copy* primitive [4]. It is intended to copy n words from a process pid_1 at address a_1 to a process pid_2 at address a_2. In the context of an operating system it is widely used to implement process management routines, as well as IPC. The correctness is justified by the instances of Theorems 4.1 and 4.2, where $f = copy$, and $p_f = p_{copy} = pid_1, pid_2, a_1, a_2, n$.

[4] Implementation and Isabelle/HOL theories containing proofs are available from www.verisoft.de.

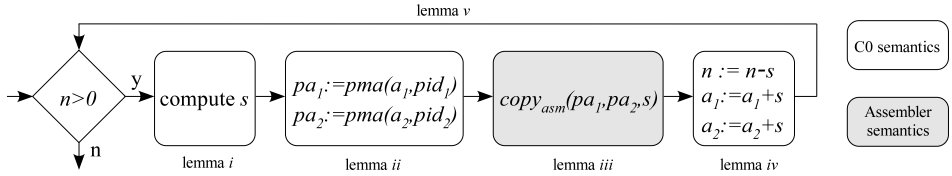

Figure 3. Algorithm and structure of auxiliary lemmas for the *copy* primitive

5.1 Algorithm

Let $copy_{asm}(pa_1, pa_2, s)$ be an assembler fragment that copies s words in the memory from a physical address pa_1 to pa_2. The algorithm behind the *copy* primitive is as follows. In a loop until n words are processed, we compute the size s of portions to be copied respecting the page borders of both processes. The crucial observation is that both pages, from and to which we copy, must be present in the physical memory. This is achieved by two consecutive calls to the page fault handler which compute physical addresses $pa_1 = pma(pid_1, a_1)$ and $pa_2 = pma(pid_2, a_2)$ and guarantee that both pages containing pa_1 and pa_2 reside in the main memory. We proceed with the copying by executing $copy_{asm}(pa_1, pa_2, s)$. The idea is depicted in Fig. 3.

5.2 Specification

The specification of the primitive has to reflect the changes on (i) the user processes *cvm.up* of the model, (ii) the concrete kernel k, and (iii) the underlying physical machine *pm*. Of course we do not specify just the same conditions in terms of each modified machine because they are interconnected through the simulation relations. We define the desired result on a sufficient level of abstraction. Nevertheless, there is a number of necessary properties, mostly of technical nature, that could be expressed only in terms of the implementation machines k and *pm*.

Effects on the Model

Let for a memory m an access to d consecutive cells starting at address a is abbreviated as $m_d[a] = m[a+d-1] \circ \ldots \circ m[a]$. The effect of the primitive execution on the model is given by the function $copy_S(cvm.up, pid_1, pid_2, a_1, a_2, n) = cvm'.up$ which updates the memory of the user process pid_2, i.e., the virtual machine $cvm.up[pid_2]$. Formally, $cvm'.up[i].m_n[a] = cvm.up[pid_1].m_n[a_1]$ if $i = pid_2 \ \wedge \ a = a_2$, otherwise $cvm'.up[i].m_n[a] = cvm.up[i].m_n[a]$.

The result of $copy_S$ is welldefined only if the preconditions $pre_{copy_S}(cvm.up, pid_1, pid_2, a_1, a_2, n)$ are satisfied. Otherwise, the same trick as with C0 machines is used. The model state space C_{cvm} is extended with a single error state which signals, in particular, that the preconditions to a primitive are not justified. The validity requirements over a model run prevent error states. The predicate pre_{copy_S} encodes formally the following: (i) the amount to be copied is reasonable: $n > 0$, (ii) we copy between different processes: $pid_1 \neq pid_2$, (iii) since memories of virtual machines are word-addressable, the addresses a_1 and a_2 are divisible by 4, (iv) process identifiers pid_1 and pid_2 lie in the interval $[1, N)$, (v) virtual machines $vm_1 = cvm.up[pid_1]$ resp. $vm_2 = cvm.up[pid_2]$ have amount of virtual memory storing resp. sufficient to store the desired portion, i.e.,

$a_x/4 + n < vm_x.ptl \cdot P$, $x \in \{1, 2\}$.

Effects on the Implementation

The intended modifications of the physical machine pm, on top of which the concrete kernel k runs are defined by the postcondition $post_{copy}(k', pm')$. First, it claims the value of the result variable of the call. Next, it describes the changes in the physical memory of the updated machine pm'. Recall that a virtual address va of a process pid is translated to the physical one by means of the function $pma(pid, va)$. Then, the changes over the physical memory are: $pm'.m_n[a] = pm.m_n[pma(pid_1, a_1)]$ if $a = pma(pid_2, a_2)$, otherwise $pm'.m_n[a] = pm.m_n[a]$.

The result is obtained only if the precondition is satisfied. The predicate $pre_{copy}(k)$ defines the indispensable demands to the C0 implementation. Basically, it speaks about parameters and results of the call and demands that: (i) the parameters $va(k, i)$, $0 \leq i < 5$ of the primitive are welltyped, evaluate without errors, storing valid values, and (ii) the result variable of the call is present in $top(k)$. We do not have any special demands to the physical machine before the call to *copy*. Nevertheless, it is worth to mention the preconditions $pre_{copy_{asm}}(pm)$ to the assembler portion $copy_{asm}(pa_1, pa_2, s)$ of the function. They are discharged when we verify the C0 prefix of the function and perform the semantics switch. The requirements comprise, among others, the following conditions: (i) $s > 0$, (ii) the addresses are divisible by 4 and bounded by the total amount of physical memory $pa_x/4 + s < TPP \cdot P$, $x \in \{1, 2\}$, and (iii) the memory regions between which we copy do not overlap: $[pa_1/4 : pa_1/4 + s) \not\cap [pa_2/4 : pa_2/4 + s)$.

5.3 Proving Correctness

In order to prove Theorem 4.1 we show the following separate lemmas for the correctness of: (i) the C0 code inside the loop up to the first call to the page fault handler, (ii) the two consecutive calls to the page fault handler, (iii) the inline assembler portion, (iv) the remaining C0 part of the loop body, and (v) the whole loop, which makes use of the four previous lemmas (cf. Fig. 3). We motivate such modularization as follows. We create separate lemmas for the items i, iii, and iv because they describe code portions in alternating semantics. The case ii is treated specially as it speaks about the properties derived from the page fault handler specification. They are used for lemma iii, but either invisible or not important in the other lemmas.

The proof proceeds by applying the respective semantics to the code statement by statement. In order to prove the loop, i.e., lemma v, we formally specify the ranking function $r(n, a_1, a_2) = i$, s.t. $r(0, a_1, a_2) = 0$ which counts the number i of the remaining loop iterations. The lemma has an inductive fashion with the step of the form $P(k, pm, i) \Longrightarrow P(k', pm', i - 1)$. Hence, its proof justifies the loop termination.

Lemma ii argues that after two consecutive calls to the page fault handler for the computation of physical memory addresses pa_1 and pa_2 both pages containing these addresses reside in the physical memory. A design requirement not to swap out the page that was swapped in during the previous run is respected by the page fault handler. However, this property *cannot* be stated directly in its specification. It is

expressed in the specification of two successive calls to the page fault handler. We deal with the problem as follows. Let $page(pa)$ denote the physical page corresponding to the physical address pa. We have a look at the eviction algorithm stated in the page fault handlers formal specification and find out the property $ev(p)$ a page p must obey in order to be evicted during the next call to the handler. The first call to the handler yields the translated physical memory address pa_1. The swapped in page is then $page(pa_1)$. We prove that $\neg ev(page(pa_1))$ holds after the first call the page fault handler. Since there is no code between the two handler calls, this property holds in the precondition to the second call for free.

Lemma iii states that the physical addresses from (pa_1) and to (pa_2) which we copy are associated with processes pid_1 resp. pid_2 and do not belong to the kernel range. This ensures kernel safety—the assembler portion does not destroy kernel by modifying its data structures—and security, for instance we do not disclose page tables by coping them to a user process. Next, lemma states that pa_1 and pa_2 do not belong to the hard disk port address range, guaranteeing that no swap data is disclosed or modified . The proof exploits lemma ii because most of these properties are inferred from the page fault handler validity (cf. Sect. 4.2.2).

The proof of Theorem 4.2 uses the functional correctness established above. The essential proof goals are the implication from the postcondition $post_{copy}(k', pm')$ to (i) the $\mathcal{B}$-relation $\mathcal{B}(f_S(cvm.up, p_f), pm', hd')$, and (ii) the kernel invariant $inv(cvm', k', pm', hd')$. The proof of the former necessarily exploits the fact that the $\mathcal{B}$-relation is preserved under the page fault handler. Since the relation is not affected by the C0 parts of the primitive—we do not write to PCBs and cannot modify the memory region beyond C0 variables—it remains to show that the $\mathcal{B}$-relation is not destroyed by the assembler fragment. Here, since we transfer data between pages residing in the physical memory formally described in lemma iii and perform the respective memory updates on the model cvm and on the physical machine pm as stated in Sect. 5.2, the relation follows. During the kernel invariant proof we examine $post_{copy}(k', pm')$ in order to determine which C0 variables and memory parts are changed by the primitive. From this the invariant is concluded.

6 Summing Up

We have shown how the problem of formal correctness of microkernel primitives is solved exploiting a novel approach to verification of C with inline assembler programs. We conclude by giving statistics and directions for further research.

Complexity

The implementation of 16 primitives consumes about 600 lines of code. We have verified 3 primitives so far—*copy*, *get_vm_word*, and *set_vm_word*—which are implemented with 100 lines. The functional correctness of the exemplary primitive (Theorem 4.1) is established in about 2K proof steps, where proofs in C0 and in assembler semantics are related as 2:1 [5]. The integration of these results into the

[5] Approximately the same proportion holds for the respective parts of the implementation.

kernel context (Theorem 4.2) requires about 5K commands using general kernel lemmas of technical nature proven in 5K steps.

Further Work: Gaining from Automated Verification

Although we used mostly interactive verification techniques there is a room for the automation. One can gain from methods of automated verification while proving the functional correctness of source code. We used the ML code generation mechanism for the proof of the microkernel source code wellformdness properties required by the C0 compiler correctness theorem. That saved about 1K proof commands. Next possible candidate for proof automation are assembler portions. Due to the relatively simple finite memory model it might be possible to obtain the values of desired memory cells by means of model checking. In order to ease the C0 part verification, one can think of a Hoare logic environment for the C0 small step semantics which will automatically generate verification conditions to be proven.

References

[1] Alkassar, E., N. Schirmer and A. Starostin, *Formal pervasive verification of a paging mechanism*, in: *14th Intl Conference, TACAS 2008, Proceedings (to appear)*, LNCS (2008).

[2] Beyer, S., C. Jacobi, D. Kroening, D. Leinenbach and W. Paul, *Putting it all together: Formal verification of the VAMP*, Intl Journal on Software Tools for Technology Transfer **8** (2006), pp. 411–430.

[3] Gargano, M., M. Hillebrand, D. Leinenbach and W. Paul, *On the correctness of operating system kernels*, in: *Proc. of the 18th Intl Conference on TPHOLs*, 2005, pp. 1–16.

[4] Heiser, G., K. Elphinstone, I. Kuz, G. Klein and S. M. Petters, *Towards trustworthy computing systems: Taking microkernels to the next level*, SIGOPS Oper. Syst. Rev. **41** (2007), pp. 3–11.

[5] Hillebrand, M., T. In der Rieden and W. Paul, *Dealing with I/O devices in the context of pervasive system verification*, in: *ICCD '05*, 2005, pp. 309–316.

[6] In der Rieden, T. and A. Tsyban, *Cvm - a verified framework for microkernel programmers*, in: *3rd intl Workshop on Systems Software Verification (SSV08), to appear* (2008).

[7] Leinenbach, D., "Compiler Verification in the Context of Pervasive System Verification," Ph.D. thesis, Saarland University, Computer Science Department (2007).

[8] Moore, J., *A grand challenge proposal for formal methods: A verified stack*, in: *Proc. of the 10th Anniversary Colloquium of UNU/IIST*, 2003, pp. 161–172.

[9] Ni, Z., D. Yu and Z. Shao, *Using xcap to certify realistic systems code: Machine context management*, in: *Proc. 20th TPHOLs* (2007), pp. 189–206.

[10] Paul, W., *Towards a worldwide verification technology*, in: *Proc. of the Conference on Verified Software: Theories, Tools, Experiments*, Zürich, Switzerland, 2005.

[11] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technische Universität München (2006).

[12] Tews, H., *Micro hypervisor verification: possible approaches and relevant properties*, electronic version is available at http://www.cs.ru.nl/H.Tews/nluug-07/hyperveri.pdf (2007).

Verified Safety and Information Flow of a Block Device

Paul Graunke

Galois, Inc.
Oregon, USA

Abstract

This work reports on the author's experience designing, implementing, and formally verifying a low-level piece of system software. The timing model and the adaptation of an existing information flow policy to a monadic framework are reasonably novel. Interactive compilation through equational rewriting worked well in practice. Finally, the project uncovered some potential areas for improving interactive theorem provers.

Keywords: multi-level security, higher-order logic, domain-specific language, monads

1 Introduction

Since system software underlies practically all applications and robust applications require robust foundations, the construction of robust system software is important. Constructing software out of small components of limited functionality keeps the complexity of any individual component manageable. The principle of least privilege dictates that designs reduce the number of components whose correct operation is critical. Formal verification of these few small components is both feasible and worthwhile. This document discusses the design and verification of one such component.

The particular system is a multi-level secure remote-mountable file server. Since the networked file server connects to multiple networks each operating at a single security level, the server integrates into the existing network infrastructure of organizations currently using (mostly) separate networks to maintain data separation.

The main theorem proved relates to information flow. Specifically, each network has an assigned label from a partially ordered set of labels. For each label, all observations made at an interface with that label depend only on inputs from interfaces whose labels are less than or equal to the label in question.

This paper also touches on a number of techniques used or hurdles overcome during the project. It describes the author's experience embedding a low-level language in higher-order logic using monads, reasoning about the embedded programs,

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

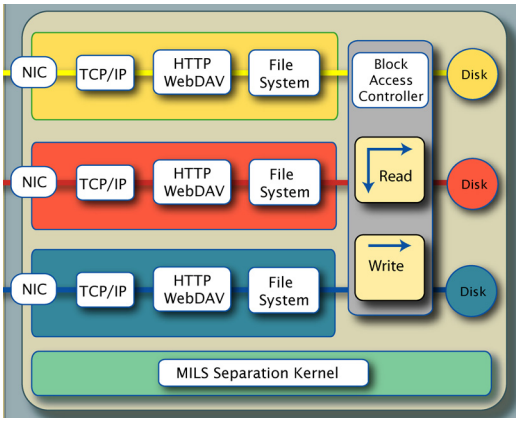

Fig. 1. Internal Architecture

and connecting those programs to an implementation. The work presents a modified formalization of non-interference more suited to programs with non-inverted control flow and weakened to allow for timing variations. Finally, it suggests a few areas where tool improvements could help future efforts of similar nature.

The rest of this document is organized as follows. Section 2 briefly presents the internal system architecture. Section 3 provides an overview of the modeling approach. Section 4 focuses on the model of the environment surrounding the software. Section 5 explains the proven properties of the system. Section 6 outlines the general proof technique. Section 7 describes some manually applied transformations in order to better match the capabilities of the theorem prover. Section 8 explains the derivation of a low-level model fairly close to the final implementation. Section 9 discusses the steps taken to produce the implementation from the low-level model. Section 10 describes property-based and model-based testing of the implementation. Section 11 discusses various improvements that may be helpful in the future. Section 12 concludes.

2 Internal Architecture

Figure 1 depicts the internal process architecture of the system. The system services each network with a front end process that consists of a user-mode network stack and a network-enabled file system. Each front end interfaces to the persistent data storage through the block access controller. Since this is the only component in the system that spans multiple security levels, it is the only user-level component requiring formal verification. Since the project relies on another organization's separation kernel supporting the GWV security policy [8], the remainder of this paper focuses on the block access controller.

3 Modeling Overview

Low-level system software involves many imperative effects including state updates, input, and output. Some behaviors of various hardware devices are at best difficult to formally specify or, even worse, explicitly documented as unspecified. The logic used for the project, namely Isabelle [11]'s higher-order logic [7], lacks direct support

for such impure operations. The project bridges this gap via a shallowly-embedded domain-specific programming language suitable for constructing low-level system software. Monads provide a convenient formalism for structuring denotational semantics so that one may program directly with the denotations [14]. Indeed the Haskell programming language pioneered this approach which is now regularly used. Section 4 further describes the monadic type, including its mutable state, I/O, and unspecified error state.

Another issue with modeling system software is that higher-order logic functions are total, while most system software does not terminate. The program transformations mentioned in section 7 (informally) transform the interactive program into a terminating reaction function surrounded by a single outermost non-terminating reaction loop. This terminating reaction function, step, then formally models the software component as a transition system.

Thus, the block access controller's step function has approximately the following interface.

step :: "config ⇒ unit m"

That is, the step function consumes a configuration which specifies statically configured details about the number of interfaces, labels associated with them, and the partial order between the labels. It then produces a monadic computation with no useful return value (i.e. of type unit). As in Haskell, even though the main function has type IO (), the intermediate computations combined with the monadic bind operator do produce useful results.

While reasoning directly in higher-order logic works well in many situations, properties about essentially imperative monadic computations can benefit from a Hoare-like logic. Since higher-order logic easily embeds other logics, this is a simple task. The predicate "prePost c p q x" consumes some static configuration information (c), a precondition (p) over the pre-state, a post condition (q) over the value produced by the monadic computation and over the post-state, and the monadic computation itself (x). The predicate roughly means that every possible run of the monadic computation starting from a start state satisfying the precondition results in a well-specified state that satisfies the post condition.

4 Environment Model

One of the challenges in formal verification is modeling the environment in which the formally modeled component executes. In this case, all interactions between the program and its environment occur by the program executing primitive non-proper morphisms of monadic type. The environment consists of hardware devices and interprocess communication channels. Since both the hardware devices and other processes operate in an asynchronous concurrent manner, the complexity of the system could increase dramatically without a carefully chosen model of time.

Figure 2 presents three groups of monadic primitives related to I/O. The first group of primitives simply report the number of input, output and DMA buffers, the number of disks, and the capacities of the disks in blocks respectively. [1] The

[1] The extra type variable σ parameterizes the state space. It can be largely ignored, although it helped eliminate frame conditions by restricting access to the state space within various computations.

numIvec :: “(nat, σ) m”
numOvec :: “(nat, σ) m”
numDvec :: “(nat, σ) m”
numDisk :: “(nat, σ) m”
diskSize :: “diskIndex ⇒ (nat, σ) m”

ivecRef :: “ivecIndex ⇒ nat ⇒ (byte, σ) m”
ovecRef :: “ovecIndex ⇒ nat ⇒ (byte, σ) m”
ovecSet :: “ovecIndex ⇒ nat ⇒ byte ⇒ (unit, σ) m”
dvecRef :: “dvecIndex ⇒ nat ⇒ (byte, σ) m”
dvecSet :: “dvecIndex ⇒ nat ⇒ byte ⇒ (unit, σ) m”
dvecBusy :: “dvecIndex ⇒ (bool, σ) m”

numPending :: “diskIndex ⇒ (nat, σ) m”
startDma :: “diskIndex ⇒ dvecIndex ⇒ bool ⇒ blockId ⇒ nBlock ⇒ (unit, σ) m”

Fig. 2. Monadic Primitives

second group of primitives read a byte from or write a byte to input output and DMA buffers or check if a DMA buffer is busy. The final two primitives check the number of DMA requests currently in flight for a given disk or initiate a new DMA transfer. The next two subsections discuss the shared memory interprocess communication model and the disk model further.

4.1 Interprocess Communication Model

The block access controller interfaces with other software processes through shared memory pages. Each shared memory page is either readable and writable by the block access controller and read-only by the other process or read-only by the block access controller and may be both read and written by the other process. The model treats each direction separately.

For input buffers an oracle function determines the value of the input. The question, however, is choosing the parameter of the input oracle. On what do the input bytes depend? They actually depend on the other processes, including what those other processes have read from the block access controller. Modeling all of those factors in detail would be inconvenient at best. Instead, a byte read is a function of the memory page being read, the offset within that page, and some notion of time. What is an appropriate notion of time? In a small step operational semantics, the number of reductions could model time. This is unnecessary since the amount by which we increase the time argument between reads is irrelevant as long as it changes. Using the total number of bytes read from any offset on any memory page would not work well. If the number of bytes read from a supposedly unobservable memory page influenced the values of the bytes read from an observable memory page, then the information flow policy would be false. Is this a legitimate problem, or a spurious flow of information due to a modeling error? Since each execution of the step function is short enough to finish within a single time slice and since none of the processes are scheduled simultaneously, the input values across all input pages remain stable throughout a reaction. Feeding the reaction number into the input oracle is a reasonable choice. This assumption that the input buffers remain

completely invariant during a reaction is a stronger assumption than actually required. Instead, the model assumes that every time the system reads a byte from an offset within a given input page it reads a (potentially) fresh value, but that value is independent of any of the other inputs or the amount of unrelated computation performed. Thus the part of the state space related to input stores a two dimensional array of natural numbers, i.e. "time array array". The input oracle has type "(time ⇒ byte) array array".

Output is somewhat simpler than input. The program can read the memory it wrote, so the model stores the output bytes on each output page as an array in the state space. It also, however, stores a history variable that records the trace of all instructions executed that wrote a byte to an output array. This may not be necessary; recording the output state at the end of each reaction would suffice. On the other hand, making a stronger claim that even if an attacker could see the order in which the block access controller wrote all the bytes to the output page during a reaction, the attacker could still not glean any information about requests from unobservable channels.

4.2 Disk Model

The main issue in modeling hard disks is the asynchronous nature of DMA. While using synchronous disk operations would have greatly simplified the system, the performance would likely suffer unacceptably. DMA, like shared memory interprocess communication, must be modeled with care in order to reason about the interleaved concurrent actions at a granularity coarser than individual memory bus cycles. The key idea is that as long as nobody is looking, the extent to which the DMA transfer completed so far remains irrelevant. The two primitive non-proper morphisms for reading a byte from a DMA buffer and for writing a byte to a DMA buffer both check if any disk is currently using that buffer for a DMA transfer. If so, the operation transitions to the unspecified error state.

There is also a distinction between a buffer not being in use vs being known to not be in use. It is at least ideologically incorrect for a program to initiate a DMA transfer and then access the DMA buffer without checking for the transfer's completion even if the transfer has actually completed. Therefore the program initiated check for DMA completion also performs the actual transfer upon completion. In effect, the trickle of concurrent DMA data transfer becomes one large synchronous and atomic transfer at a later point in the program's execution.

To support this model of DMA, the model of each disk contains two pieces of actual state and two fictitious pieces of state used for modeling time. The first actual piece of state is an array of disk blocks where each block is an array of bytes. This is the state one normally thinks of as the state of a disk. In addition, each disk has a queue of pending DMA requests paired with time stamps. Also, for the sake of modeling timing, the disk maintains the history of all requests sent to the disk and the total number of times the program has checked for DMA completion.

Initiating a new DMA transfer checks that all the indices are in bounds and that the buffer is not used by any other active transfers. It then wraps its arguments up in a pending request structure. Next it passes this structure, the disk number, the

history of all prior requests to the disk, the current pending requests, and the total number of completion checks to a disk timing oracle. The timing oracle returns a completion time. The model adds this time to the maximum completion time for requests already in progress in order to assure monotonicity, which the advanced host controller serial ATA interface guarantees about hard disks.

5 Policy

A central goal of the system—and of the block access controller—is the prevention of information leakage. A significant lemma necessary for reasoning about other properties is that the system behavior is well specified. The proof regarding information flow then relies on the facts established during the safety proof.

5.1 Safety Policy

The safety property states the invariance of the goodState predicate with respect to the step function.

```
prePost c (goodState c) (λv. goodState c) (repeat n (step c))
```

The use of prePost also asserts that the result is not the unspecified state. If running a monadic computation does not result in the unspecified error state, then none of the intermediate computations did either. This is due to the monadic bind operation propagating the unspecified state, which is easy to prove. Demonstrating that none of the shallowly embedded language constructs can catch the abortive nature of an unspecified computation requires an inspection of all the uses of the primitive monadic data constructor to check for catch-like non-proper morphisms. While inconvenient, at least the task remains relatively straightforward.

The goodState predicate contains a number of invariants. Arrays must be the proper size for the given configuration. Queues of partially processed requests must have resulted from legitimate requests. As an example, for each in-flight request to send a data block to an interprocess communication channel the receiving channel must be authorized to read that data block.

As an aside, the sequential composition rule for Hoare logic (adjusted for a monadic setting) offers a cleaner alternative to disjunctive invariants. A disjunctive invariant is typically applied uniformly throughout the program, but the first conjunct of each disjunct serves as a guard based on an encoding of the control flow as a data value. They often have a form similar to the following.

```
P x = atProgramPointA x ∧ P₁ x ∨ atProgramPointB x ∧ P₂ x
```

Instead of encoding the control state of the program as data, the following Hoare-style rule for reasoning about the monadic bind operation directly supports different predicates for different control flow points.

```
lemma prePostBind
: "⟦prePost c p q x; ∀v. prePost c (q v) r (y v)⟧ ⟹ prePost c p r (x >>= y)"
```

This prePostBind rule includes an arbitrary predicate q over the intermediate

state. Thus, the pre and post conditions p and r might both be goodState while q may be some other predicate. For example, two queues may normally need to have equal length, but immediately after updating one queue and before updating the other to match the lengths differ by one. Thus, the logic handles different invariants at different points in the control flow without encoding the control flow as a data value.

Separating the concerns of basic well-defined behavior from information flow worked well in practice as this kept each proof more manageable. Merely starting the information flow proof with the assumption that the behavior was well-defined, however, was inadequate. The facts established during the safety proof at each intermediate point in the step function's execution contained useful properties needed during the information flow proof. Inserting assertions into the code helped transfer these facts to the context of the next proof. The operation assert p either leaves the state unchanged (and returns the unit value) if the predicate p holds for the program state; otherwise it transitions to the undefined state. Discharging the extra assertions is trivial since the inserted predicates readily follow from facts available during the safety proof. The assertions strengthen the safety theorem so it also implies all the intermediate properties asserted throughout the code. This transfers the asserted facts to the information flow proof.

5.2 Information Flow Policy

The information flow model of this section is essentially Denning's lattice-based model [3], but with the weaker requirement of a partial order rather than a lattice. Greatest lower bounds are useful for combining users or consumers of data, but the file system never does this. While least upper bounds are useful for assigning labels to combinations of data with different labels, the file system never initiates the combination of data items. Instead it only combines data at the request of a client; for example, the server generates directory listings which refer to files of different labels. Although the client's label may be greater than the least upper bound of the labels in the directory listing (if such an upper bound exists), the client's label suffices.

Goguen and Meseguer's notion of (transitive) non-interference [6] essentially captures the desired property and is widely accepted as a reasonable policy. Intuitively two imaginary copies of the system run in parallel where one copy receives only a subset of the inputs. The system respects Denning's notion of information flow if for each label the outputs of both copies look the same from the viewpoint of an observer associated with that label when one copy of the system sees only inputs from labels less than or equal to that of the observer. That is, the output was not influenced by allegedly unobservable inputs.

The formulation of a policy, however, depends on the representation of the system to which the policy applies. While most existing notions of non-interference assume an external stream of labeled events that repeatedly prod the system into action, this work's model of computation more closely matches that of a traditional program. That is, the program executes without any external prodding and of its own volition consumes input when it wishes. This meshes well with the interprocess communication mechanisms and scheduler provided by the underlying real-time

```
nonInterference :: "config ⇒ σ machine ⇒ (α, σ) m ⇒ bool"
"nonInterference c s x
 ≡∀obsL w t
   .obsL < numLevels c
    ⟶ w ∈ input-set c
    ⟶ (∃u. outputEq c obsL
                (runM x (purge c obsL w) u s)
                (runM x w t s))"

theorem stepNonInterference
: "⟦goodConfig c; goodState c s⟧
   ⟹ nonInterference c s (repeat n (step c))"
```

Fig. 3. Notion of Noninterference

kernel. Another difference is that the step function processes multiple requests from multiple sources with different labels.

These differences in the computational model require adjustments to the policy, as shown in figure 3. Rather than dropping input actions that must remain unobservable, the purge function filters the input oracle so any input bytes read from unobservable sources are zero. Conveniently, zeroing inputs to steps rather than skipping steps fits better with the simple scheduling algorithms found in real-time kernels and also eliminates stuttering from the proofs.

In addition to adjusting the input oracle, the policy also adjusts the disk timing oracle. The cause for the adjustment is that requests to read data causes a delay in the processing of other disk requests. While the deployed system may rely on pragmatic solutions to cover this delay, the model accommodates the issue by weakening the policy. The purged execution trace may chose any arbitrary disk timing oracle. Thus, the theorem says that any differences in system behavior could potentially be caused by the disk drive running at an unusual speed.

The theorem at the bottom of figure 3 states that the monadic computation that runs the step function an arbitrary n times supports the non-interference property at the top of the figure for the given initial state. The non-interference property states that for any observer label obsL and input oracle w and timing oracle t if the label is valid and the input oracle is valid, then running the monadic computation from the start state on both the filtered and unfiltered input oracles sends the same trace of outputs to observers with the given label.

6 Proof Technique

The structure of the information flow proof is fairly typical, although it requires some adjustment for the monadic framework. This section describes the proof structure, the similarities to Ohëimb's variation [13] of the unwinding lemmas, and some key distinctions.

The predicate outputEq in figure 3 is not inductive. Thus, the first step in the proof is to strengthen the induction hypothesis. A stronger relation viewEq insists not only the outputs are equal but the pieces on internal state which may eventually

indirectly influence the output must also be equal. Proving that viewEq is in fact a stronger relation than outputEq is traditionally called output consistency. Defining viewEq as the conjunction of outputEq and a predicate on internal states makes this proof trivial.

The viewEq relation is an equivalence relation. The fact that it is reflexive starts the induction over two traces of states by asserting the initial state is viewEq to itself. Higher-order logic supports Harrison's technique of proving that for any equivalence relation R, R x y = (R x = R y). This changes a custom equivalence relation into Leibniz equality, which Isabelle's simplifier can then use to perform equational rewriting with viewEq and automatically make use of symmetry and transitivity.

The bulk of the work is proving that stepping both traces one step preserves the viewEq relation. There are two common cases to consider. Either the monadic action affects only state that is observable and does not depend on any unobservable state or the action affects only unobservable state. The first corresponds to Ohëimb's weak step consistency while the latter corresponds to local respect. One difference is that while action-based frameworks case split on the level associated with an external action, in this system the case split depends on (some argument to the function that generates) the monadic computation or some piece of the state space or both.

For the first case where the action modifies observable state, the proof decomposes the lemma over the monadic bind operation. The key is that in the observable case, the values produced by the first computation in each trace are Leibniz equal. This causes the application of the second argument of the bind operator to produce equal computations in each trace. This enables the proofs about the two subcomputations of the bind operation to combine into a proof about the whole computation.

For the second case where the updated state is not observable, the proof reduces the reasoning to focus on only a single trace. The updated state in each trace is viewEq to the state before taking the step. Thus, the equivalence of the final states of two traces follows from transitivity and the equivalence of the traces' starting states. Instantiating quantifiers is easier when there is only one expression in the assumptions list of the proper type, so reasoning about each trace independently improves automation.

7 High-Level Model

The process of designing the high-level model of the step function mentioned in section 3 is somewhat interesting, but largely out of scope. This section highlights a few techniques.

The initial design consisted of a direct-style program that reads a request, performs the appropriate blocking I/O operation, waits for the response, and returns the result. This program suffered from several problems. First, higher-order logic cannot directly state the desired system property in terms of this interface. Second, the program processes only one request at a time.

Inverting the control flow [10] through continuation passing style [12] solves both problems. Each formerly blocking I/O operation instead aborts the current contin-

uation and invokes a callback when the I/O completes. To handle multiple requests, a continuation delimiter [4] surrounds the loop body. The delimited continuations capture and abort only the loop body rather than the entire remaining computation. Thus, the program immediately continues with the next iteration which initiates more asynchronous I/O requests. This, in essence, introduces cooperative threading.

Another benefit of inverting the program is that the I/O moves to the end of the function rather than the middle. This aligns the boundaries of the step function with the process's time slice. Thus, the policy is in terms of observations made when the program is blocked for I/O and other processes are running rather than at some arbitrary point.

8 Low-Level Model

From the high-level model of the step function, judicious use of Isabelle's equational rewriting produces a lower-level model. While the high-level model is already close to source code in a language with good support for functional programming, features such as dynamic memory allocation present challenges for languages typically used in systems programming.

Transforming the representation of the step function relies on a synonym for equality "eqP $\equiv$ op =" which delays the instantiation of schematic variables. Specifically, folding the definition of eqP in the lemma "step = ?x" prevents the simplifier from immediately solving the goal by reflexivity.

A variety of optimizations are readily available "for free" just because the development occurs within a theorem prover. Unfolding constant definitions performs both function inlining and constant propagation optimizations. The only mild issue with inlining is that some definitions require η-expansion before the simplifier will use them. Rewriting with the monadic laws are almost free, since the proofs are unproblematic and should be proven anyway.

Although assertions made facts proven during the safety proof available during the non-interference proof, they serve no purpose in the implementation. Isabelle easily removes the assertions via rewriting. Under the assumption that running an assertion does not result in the unspecified error state, running the assertion is equivalent to running return (). Isabelle can use this lemma along with lemmas that assertion removal is compatible with the structure of the program syntax as elimination rules in order to perform the desired syntax-directed program transformation.

The high-level representation of the step function contained two sources of dynamically allocated intermediate data structures, which Isabelle deforested in different ways. The first was relatively straight forward, while the second involved more effort.

The first structure represents incoming requests from the interprocess communication channels. The program bound the result of parsing the request to a variable and then applied a function to the variable to process the request. The parser consists of several nested conditionals which test for each possible variant of the request structure. Each leaf of the tree of conditionals constructs the appropriate request

```
enqueueCont :: "label ⇒ cont ⇒ unit m"
dequeueCont :: "label ⇒ cont option m"
```

Fig. 4. Enqueue and Dequeue Operations

variant. The processing function first case splits on the request variant and then takes appropriate action. Inlining both functions and then distributing the processing function across the parser's conditionals replicates the processing function n times where n is the number of variants. In this case, however, each replicated instance of the case split is immediately applied to a (fully saturated) data constructor. Each of the n case statements then reduce to a single branch which is on average $1/n^{th}$ the size. Thus, the transformation decreases code size rather than increasing it and also eliminates the dynamic construction of compound data.

The second dynamically allocated data structures represent work left to be done after a DMA request completes. A naïve implementation of the high-level model would allocate such a structure and enqueue it just before sending a DMA request to a disk. The processing of the structure occurs later when the DMA completes; the structure construction and the structure destruction are in some sense far apart—even in different reactions.

To deforest the structures, consider the non-proper morphisms that operate on the state space within the monadic type. Figure 4 shows the approximate type signatures of the high-level model's operations for enqueuing and dequeueing these (defunctionalized partial continuation) structures. This interface to the queue data structure suffers from a problem similar to that of the generic state monad mentioned by Chen and Hudak [1]. Specifically, the state representation inside the monadic datatype contains references to values passed in from the outside. [2] Since the interface to the monadic datatype fails to enforce linearity, the operations do not model an imperative implementation. Transforming the interface eliminates this difficulty.

The transformation to the monadic state's interface involves rewriting the model with the new queueing operations shown in figure 5. The top portion of the figure defines new operations in terms of the old ones. The lower portion proves alternate definition-like equations for the old operations in terms of the new ones. Substituting the new equations for the old operations throughout the program produces a new program representation that uses only the new interface. Inlining the original definitions for the queueing operations into the function bodies of the new operations updates the interface exposed by the monad.

This new interface to the state's queues exposes only the components of the structures, not the structures themselves. There is only ever one reference to each structure, namely the reference from a node of the queue. The linear nature of the references ensures the faithfulness of an implementation based on imperative updates. Thus, the transformation eliminates the need to dynamically allocate new structures when enqueuing.

While linearity eliminated one source of dynamic memory allocation, the transformation introduced another. The wrapper equations in the lower half of figure 5

[2] Returning a reference to a value also poses problems.

```
enqueueContRead :: "label ⇒ ovecIndex ⇒ dvecIndex ⇒ label ⇒ nat ⇒ unit m"
"enqueueContRead qLabel ovecI dvecI responseLabel responseOffset
 ≡ enqueueCont qLabel (ContRead ovecI dvecI responseLabel responseOffset)"

enqueueContWrite :: "label ⇒ nat ⇒ unit m"
"enqueueContWrite label responseOffset
 ≡ enqueueCont label (ContWrite label responseOffset)"
```

```
queuePeekCont :: "label ⇒ cont option m"
"queuePeekCont l
 ≡ do tag ← queuePeekContTag l
      ;if readTag = tag
         then do ov ← queuePeekContReadOutBuf l
                 ;dv ← queuePeekContReadDvec l
                 ;reqL ← queuePeekContReadReqLevel l
                 ;i ← queuePeekContReadReqSlot l
                 ;return (Some (ContRead ov dv reqL i))
         else if writeTag = tag
                then do reqL ← queuePeekContWriteReqLevel l
                        ;i ← queuePeekContWriteReqSlot l
                        ;return (Some (ContWrite reqL i))
                else return None"

lemma dequeueCont-def2
: "dequeueCont l
   = do k ← queuePeekCont l
        ;dequeueDropCont l
        ;return k"
```

Fig. 5. Modified Enqueue and Dequeue Operations

all allocate data structures. The location of the allocation, however, moved to a different portion of the code and occurs in a different reaction. In fact, by moving allocation sites from enqueuing to dequeuing, the allocations are right where the processing—and hence destructuring—of the data occurs. Isabelle's equational rewriting can now easily deforest the data structures.

9 Implementation

While proving that a model of a system enforces a desired policy eliminates certain design flaws, it does not guarantee that a deployed implementation behaves accordingly. For this, the implementation must correspond to the model of the program and the library code must properly implement the primitive monadic effects. This section describes the connection between the model and the source code. Section 10 discusses the primitives.

Isabelle's meta language serves as a foundation for code generation. The meta language—which is normally used to write tactics—can inspect the abstract syntax trees representing the higher-order logic formulae and is also a general purpose

```
int main()
{initState();
 for (;;){
#include "step.c"
}}
```

Fig. 6. Implementation Main

programming language. Since the low-level model of section 8 already looks like imperative-style source code, the code generator simply traverses the abstract syntax tree via pattern matching and recursion. At then end of the traversal the code generator simply writes the resultant string to a file, which an existing C compiler compiles. While C is not ideal for building reliable software, constructing a verified or verifying compiler all the way to machine language was outside the scope of the project. This may cause difficulties for an evaluation, since some organizations (such as the United States Federal Aviation Administration) who perform evaluations of C code require that the code to be tested using multiple compilers and that the developers disassemble the object code and explain how it relates to the compiler's input.

The generated implementation corresponds directly to the low-level model of the step function. Even with the library that implements the monadic primitives, this is not a complete implementation. A small amount of code must setup the initial state and a loop must repeatedly call the generated step function. Figure 6 is essentially the main program.

10 Testing

What is the relation between proof and testing? Some people question the value of formal methods, while others assume that proving software correct "once and for all" eliminates the need for testing the software on specific inputs. This section mentions the relation between testing and mathematical proof, two rationale for testing, and the specific testing methods employed by the project.

Mathematical proof is based on deductive reasoning—deducing more complex facts from presumably simpler ones. While deductive proofs provide value by both reducing the complexity of and clarifying the assumptions, assumptions remain. For the development presented so far, the most significant remaining assumption is that the abstract machine model of the non-proper morphisms matches the behavior implemented by the hand-written C code, the C compiler, the separation kernel, the microprocessor (hardware and microcode), and the peripherals. This layer requires validation testing to ensure it matches the model.

Since testing is typically less expensive than formal proof, testing establishes an acceptable level of assurance for less critical properties. For example, if the system suddenly became as unresponsive as a brick then it would still succeed in its primary mission of not leaking information.[3] This behavior, however, would likely cause consternation for the users. Thus, the tests cover several basic correctness

[3] The relative importance of these properties reflects the biases of the author's particular clients.

properties. The correctness tests are unlikely to pass if the abstract machine is broken; hence they cover validation testing also.

Haskell's QuickCheck [2] tests extra assertions about the implementation. Although the tool is designed to test Haskell programs, the test suite uses Haskell's foreign function interface to make interprocess communication requests that interact with the block access controller implementation. By stating the property in Haskell that writing an arbitrary block of arbitrary bytes and then reading it produces the same block of bytes, QuickCheck automatically generates many randomized requests and tests the implementation. Quickcheck also tests the property that concurrently writing blocks with identical arbitrary bytes and reading blocks from a different interface with a greater label results in reading blocks with identical bytes.

In addition to this property-based testing, the test suite also performs model-based testing. For this effort another model written in Haskell captures just the functional behavior of the system. This model simply represents the disk as a finite map of blocks of bytes, with read and write requests immediately performing the corresponding actions (i.e. no delay due to DMA). A QuickCheck assertion states that arbitrary sequences of arbitrary read and write requests processed by the implementation produce the same results as processing those same requests in the Haskell model.

11 Future Work

The experience with this project was overall quite positive, but room for improvement remains. The integration between the testing and verification tools could have been tighter. Model to implementation correspondence issues remain. Finally, this section suggests a research challenge in modeling reactive systems.

While the primary model is in Isabelle, the testing model is in Haskell. This involved recreating another (simpler) model in Haskell by hand. Eliminating this step and testing directly from Isabelle would be preferable, both because of work reduction and also because it would strengthen the claim that the model was tested. The Isabelle theorem prover includes a method named quickcheck which, like Haskell's QuickCheck, randomly generates test vectors. Isabelle, unlike Haskell implementations, does not include impure features such as foreign function interfaces which the implementation testing relies upon. It may be possible to use Haftmann's code generator for Isabelle [9] to produce a matching Haskell model from the Isabelle model, however, this was not available at the time.

The connection between the model and implementation remains less than ideal. For one, the code generation automatically produces C code that looks syntactically similar to the low-level model, but this is merely a syntactic translation. All the claims about issues such as the store being linear remain informal. Producing machine code for a processor with a publicly available formal model, such as the ARM processor [5] would improve the correspondence. A verifying or verified compiler for a low-level language would ease the automation of such a task. Finally, the model used natural numbers for unsigned integers. This resulted in an numeric overflow error in earlier implementations. Modeling machine word arithmetic without Is-

abelle's recently available machine word arithmetic library appeared prohibitively painful.

The continuation passing style and defunctionalization transformations should have been done either by a compiler or at least in a theorem prover rather than by hand. If Isabelle could express properties about not only top level functions but also about implicit continuations, the project could have potentially modeled and reasoned about the program in direct style.

12 Conclusion

In summary, it is feasible to use modern interactive theorem provers such as Isabelle to formally prove properties about low-level system software. Higher-order logic is well-suited for embedding both other programming languages via monads as well as logics customized for reasoning about the language's constructs. Interactive compilation through equational rewriting provides a nice mix between the control of hand optimization and the convenience, maintainability, and reliability of fully automatic optimization. Our notion of non-interference supports programs with non-inverted I/O and allows for variations in the timing behavior of peripherals.

Acknowledgments

John Matthews deserves credit for convincing the author to use the Isabelle theorem prover and patiently serving as a mentor in its use. His numerous discussions regarding the project were also invaluable. The author also thanks Dylan McNamee and the anonymous referees for their helpful comments on earlier drafts. Without the support of Galois project management, especially Laura McKinney, and the support of the project's sponsors this work would not have been possible. Thanks are also due to the creators and maintainers of the Isabelle theorem prover as well as its surrounding community.

References

[1] Chen, C.-P. and P. Hudak, *Rolling your own MADT - a connection between linear types and monads*, in: *ACM Symposium on Principles of Programming Languages*, 1997, pp. 54–66.

[2] Claessen, K. and J. Hughes, *Quickcheck: a lightweight tool for random testing of Haskell programs*, in: *ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, 2000, pp. 268–279.

[3] Denning, D. E., *A lattice model of secure information flow*, Communications of the ACM **19** (1976), pp. 236–243.

[4] Felleisen, M., "The Calculi of #v-CS Conversion: A Syntactic Theory of Control and State in Imperative HigherOrder Programming Languages," Ph.D. thesis, Indiana University (1987).

[5] Fox, A. C. J., *Formal specification and verification of ARM6*, in: *International Conference on Theorem Proving in Higher Order Logics* (2003), pp. 25–40.

[6] Goguen, J. A. and J. Meseguer, *Security policies and security models*, in: *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[7] Gordon, M. J. C., *HOL: A machine oriented formulation of higher order logic*, Technical Report 68, University of Cambridge Computer Laboratory (1985).

[8] Greve, D., M. Wilding and W. M. Vanfleet, *A separation kernel formal security policy*, in: *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.

[9] Haftmann, F. and T. Nipkow, *A code generator framework for Isabelle/HOL*, Technical Report 364/07, Department of Computer Science, University of Kaiserslautern (2007).

[10] Jackson, M. A., "Principles of Program Design," Academic Press, 1975.

[11] Paulson, L. C., *Isabelle: The next 700 theorem provers*, Logic and Computer Science (1990), pp. 361–386.

[12] Strachey, C. and C. P. Wadsworth, *Continuations: A mathematical semantics for handling full jumps, technical monograph PRG-11*, Technical report, Oxford University Computing Laboratory, Programming Research Group (1974).

[13] von Oheimb, D., *Information flow control revisited: Noninfluence = Noninterference + Nonleakage*, in: P. Samarati, P. Ryan, D. Gollmann and R. Molva, editors, *Computer Security – ESORICS 2004*, LNCS **3193** (2004), pp. 225–243, http://ddvo.net/papers/Noninfluence.html.

[14] Wadler, P., *The essence of functional programming*, in: *ACM Symposium on Principles of Programming Languages*, 1992, http://homepages.inf.ed.ac.uk/wadler/topics/monads.html.

Reliable UML Models and Profiles

Kirsten Berkenkötter

Faculty of Computer Science
University of Bremen
Bremen, Germany

Abstract

Formerly, models have been used mostly in design and documentation. MDA and its surrounding techniques put them into the center of the software development process as the platform-independent model serves as foundation for tasks such as platform-dependent code generation or testing. Obviously, unambiguous models are crucial for the successful accomplishment of these tasks. The UML as the most popular modeling language is not able to ensure this which delegates the validation of models to further tasks. Our goal is to improve this situation by making models reliable as it is neither likely that another modeling language will displace UML in the near future not that a new - improved - UML version will be adopted soon. We reuse the existing OCL-based static semantics of UML and strengthen them by rectification and extension. As a result, the structural soundness of class and object diagrams is automatically ascertained and model-based tasks can be smoothly performed afterwards. Our approach supports the usage of profiles as long as these specify their static semantics on OCL. We show this by an example taken from the railway control systems domain. Behavioral soundness is not checked as we believe that it is not desirable to define one concrete behavioral semantics for UML as different application domains require different semantics at least in details.

Keywords: UML, OCL, MDA, Profiles, Validation

1 Introduction

During the last few years, Model-driven Architecture (MDA) [15] gained more and more importance in software development. This includes attended techniques such as Model-driven Development (MDD) or Model-based Testing. The main idea in MDA is the platform-independent model (PIM) that serves as source for all other activities, e.g. transformation to platform-specific models (PSM), code generation, validation, verification, or testing. Hence, the model becomes the center of the software development process.

One of the reasons for the success of MDA is the increasing popularity of the Unified Modeling Language (UML) [17,18] that consolidates a variety of modeling techniques. The thirteen diagram types and numerous modeling elements of the current version UML2 provide means to model all kinds of software systems independent from scale and domain. The UML is accompanied by the Object Constraints Language (OCL) [16] that allows to navigate in a model and formulate

[1] Email: kirsten@informatik.uni-bremen.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

constraints on it and several other helpful mechanisms. One of them - the profile mechanism - is paid a lot of attention since the standardization of UML2 as it allows to tailor the UML to a specific domain.

New ideas in software development call for new modeling strategies. This holds especially for the UML as a successful application with respect to MDA makes new demands on the language. At the moment, UML is a loose compound of modeling techniques that lack formal semantics. Static semantics are at least partly formally defined by OCL constraints. Behavioral semantics and more static semantics are defined in natural language that is claimed to be precise. Nevertheless, lots of ambiguities are introduced. To give an example, Fecher et. al. list 29 uncertainties in the definition of UML statemachines [11].

The imprecise definition of UML is partly intended as the developers did not want to fix all semantic details to allow the application of UML for all kinds of software systems. Semantic variation points offer the possibility to adapt the UML to each domain. This is sensible as each application area has its own needs. Even if UML as a broad approach is used, developers tend to used domain-specific semantics to interpret their models. This is one reason for the success of profiles.

A completely different point is static semantics that defines the well-formedness of a model. Albeit the behavioral interpretation of models differs from domain to domain, a sound structure is needed as foundation since behavior is always related to structure in UML. Furthermore, transformations and code generation are tasks that need an unambiguous structure as source. Currently, the static semantics of UML that have been formalized are not sufficient to ensure sound models which delegates the validation of models to the tools that process them. This situation is in particular disappointing with respect to models that use profiles as each profile may define additional static semantics that have to be considered.

Approaches to formal semantics for UML tend to focus on behavioral semantics as e.g. in [11]. It is time to turn attention also on the structural well-formedness of UML models. We tackle this problem with the help of OCL as suggested in [13]. UML provides a set of OCL constraints that form the basis of static semantics. Unfortunately, many constraints are erroneous as e.g. listed in [2] or only defined in natural language. The first ones must be corrected, the latter ones formalized. Furthermore, the lengthy paragraphs intended to describe the behavioral semantics in natural language have to be carefully analyzed to detect hidden static semantics. The resulting set of OCL constraints can be automatically validated with the help of the tool USE [1,23] for class and object diagrams. In this way, we gain a reliable model that can be used by all further applications that use the model as reliable input. It is also possible to extend this process to UML profiles if these specify their static semantics in OCL [4]. We document this approach with an example profile designed for the development of railway controllers.

The paper is organized as follows: Sec. 2 presents the current degree of formalization of the static semantics of UML. After that, we show in Sec. 3 how these semantics can be strengthened by analyzing the UML specification and formulating constraints. Sec. 4 deals with the extension to profiles, while Sec. 5 handles the automated validation process. We conclude with an outlook to future work and a discussion in Sec. 6.

2 Some Notes on Syntax and Semantics of UML

The specification of a (modeling) language is structured in two parts - *syntax* and *semantics* [8]. Syntax is split into abstract syntax that defines the elements of the language and concrete syntax that provides a usable notation for users. Semantics give meaning to a language. On the one hand, these are *static semantics* that define the correct composition of syntactic elements to programs or models and *behavioral semantics* that define their runtime properties.

This separation of concepts in language design is not mirrored in the UML specification documents. Abstract syntax is given in diagrams while concrete syntax is mostly given in textual descriptions combined with figures for each modeling element. The combination of modeling elements to diagrams is described in separate sections afterwards. Static semantics are split into OCL constraints, constraints in natural language, and partly in semantics sections given for each element, mixed with behavioral semantics. Variation points have been introduced to allow different interpretations of modeling concepts due to the fact that UML has the ambition to model all kinds of systems, independent from domain and scale.

Since its first standardization in 1998, the degree of formalization of UML semantics has been fervently discussed. The developers of UML argued - and still argue - that natural language is sufficient for the description of UML semantics. A formal specification of the language *"would have added significant complexity without clear benefit"* [17]. It is also argued that *"currently, the semantics are not considered essential for the development of tools; however, this will probably change in the future."* [17]. This point of view is underlined by the fact that tool compliance to the UML standard is only defined in ways of abstract and concrete syntax but not semantics [17,18]. A tool is UML-compliant even if e.g. the interpretation of state machines is different from the one in the UML specification.

Contrary, critics argue that a certain degree of formality is definitely needed for several reasons: clarity, expendability, interoperability [10]. It is neither desirable that different persons interpret the same model differently nor that tools do the same; especially if verification or code generation is performed. This does not mean that every detail in UML shall be explicitly specified. More convenient would be a solid foundation and a well-defined extension mechanism that allows to tailor the UML to specific domains. Numerous approaches address formal (behavioral) semantics for UML, e.g. denotational [6], by Z [7], or process algebra [12]. Most of these have in common that they focus on a subset of UML, often inspired by later usage in a specific domain like real-time systems [9]. Static semantics seem to be a poor cousin of behavioral ones as they are not discussed in detail.

Some of this criticism has been regarded in UML2. The extensibility of UML has been increased and is better specified. Abstract syntax is described quite well in diagrams, but concrete syntax is still nebulous. It is still unclear, which elements can be used in which kind of diagram, as only recommendations are given. Furthermore, semantics are still informal and mostly given in natural language, even if OCL has been used more frequently than in older language versions. As many OCL constraints are erroneous [2,22], their expressiveness is dubious. The result is that the use of modeling elements in diagrams differs from tool to tool just as semantics

and their interpretation e.g. for code generation.

Profiles add significantly to this problem due to their ability to define more static semantics. These further constraints on the composition of models belong to the metamodel level and not to the model level. A tool that intends to guarantee valid UML models must be able to check both standard and profile-dependent static semantics. This requires formalization, e.g. with OCL as already done for some constraints in UML. Currently, only few tools are able to check OCL constraints on the model level let alone on the metamodel level.

Our goal is to overcome this deficiency by a validation that includes both the static semantics of UML and those of each profile. This demands an improvement of the current static semantics of UML and a formalization of static semantics in OCL for each profile. With respect to UML, we concentrate on the frequently used modeling elements of class and object diagrams as these are available in all kinds of tools. The approach can be extended afterwards to more modeling elements. As already mentioned above, we believe that each domain has its on demands on behavioral semantics just as the need for different tools for code generation, transformation, verification, automated test case generation, etc. Model validation with profile support will reduce the development time for such domain-dependent tools significantly as the model can be assumed reliable.

3 Well-formedness of UML Models

For UML, an appropriate means to define static semantics is OCL as we can navigate in the model and perform checks automatically, e.g. with the tool USE (see Sec. 5). In the UML specification documents, each modeling element has a constraints section, where OCL is partly used. Unfortunately, these constraints are often not flawless and definitely not complete. Some static semantics are hidden in the semantics section of each modeling element that in fact should contain behavioral semantics; some - that are obvious - are not mentioned at all. To improve this situation, the following steps have to be taken: (a) rectify mistakes in existing constraints, (b) formalize constraints in natural language, (c) identify static semantics in descriptions and formalize them, (d) identify missing parts.

Rectification of Erroneous OCL in UML

The reasons of errors in OCL constraints in the UML specification range from simple syntactic problems to inconsistencies with respect to the abstract syntax or misused types [2]. To give an example, we look at the classes *Association* and *Property*, defined in the *Kernel* package of UML [18]. In Fig. 1, a small excerpt of the UML metamodel that includes these classes is shown. The first problem here is related to abstract syntax. Recall that if an association end has no name, its name is the same as the class at this end with the first letter in lower case. This means that *Property* has two association ends named *association* so they are not distinct. We will use *_association* as name of the yet unnamed association end at the bottom of Fig. 1.
Several OCL problems occur with respect to this small UML excerpt, some of these are listed below:

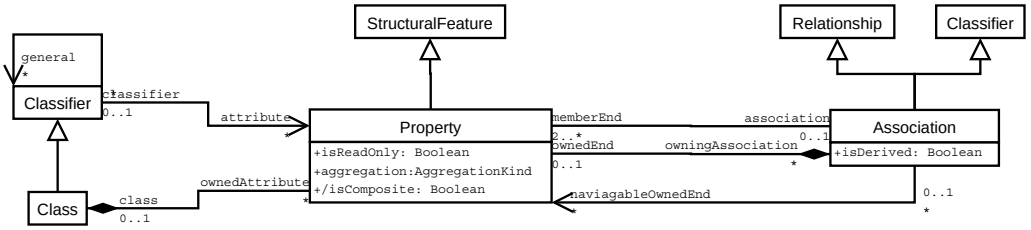

Fig. 1. Excerpt from UML metamodel - classes

- As a *Classifier*, each *Association* can be specialized. In this case, the number of ends should be the same. This is formalized as follows:

```
context Association
inv:
self.parents()->forAll(p |
  p.memberEnd.size() = self.memberEnd.size())
```

  Note that *parents()* returns the set of direct parents of the classifier. This OCL constraint has to be corrected for two reasons: (a) memberEnd is an ordered set whose function *size()* must be called with the '->' operator; (b) operation *parents()* returns a set of type *Classifier*, therefore a cast must be performed.

```
context Association
inv:
self.parents()->forAll(p |
  p.oclAsType(Association).memberEnd->size() =
    self.memberEnd->size())
```

- Further, an *Association* with more than two ends, must own all its ends:

```
context Association
inv:
if memberEnd->size() > 2 then
  ownedEnd->includesAll(memberEnd)
```

  Again, there is an error as each *if-statement* in OCL has an *else* branch and ends with an *endif*. A correct version would be:

```
context Association
inv:
memberEnd->size() > 2 implies
  ownedEnd->includesAll(memberEnd)
```

- With respect to class *Property*, a constraints states that only a navigable property can be marked as *isReadOnly*:

```
context Property
inv:
isReadOnly implies isNavigable()
def:
isNavigable() =
  not classifier->isEmpty() or
  association.owningAssociation.navigableOwnedEnd->includes(self)
```

Obviously, the operation *isNavigable()* is not correctly defined, as an *Association* does not have a property *owningAssociation* as we can see in Fig. 1. Instead, a *Property* can directly refer to this attribute:

```
context Property
def:
isNavigable():Boolean =
  not classifier->isEmpty() or
  owningAssociation.navigableOwnedEnd->includes(self)
```

Formalization of Constraints in Natural Language

In the same excerpt of the metamodel, we can find an unformalized constraint for an *Association*:

- `When an association specializes another association, every end of the specific association corresponds to an end of the general association, and the specific end reaches the same type or a subtype of the more general end.`

  Naturally, there is no reason that this constraint cannot be formalized, as we see below:

```
context Association
inv:
self.memberEnd->iterate(
  e:Property;
  res : Boolean = true |
  res = self.parents()->forAll(p |
    p.oclAsType(Association).memberEnd->exists(e2 |
      (e = e2 or
       e.redefinedProperty->includes(e2))
      and
       e.type.oclAsType(Classifier).general->includes(
        e2.type.oclAsType(Classifier)))))
```

Identification of Static Semantics throughout the Specification

An attentive reading of the UML specification brings hidden static semantics to light. Throughout the document, lots of requirements of models can be found that may be formalized. One example is the class *Classifier*. In the semantics section of this modeling element, we find the following claims:

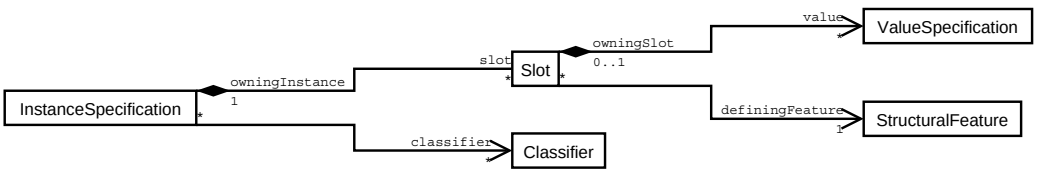

Fig. 2. Excerpt from UML metamodel - instances

- `An instance of a specific Classifier is also an (indirect) instance of each of the general Classifiers.`
  Obviously, this is not a requirement for a classifier, but for its instance that is specified by *InstanceSpecification* as we can see in Fig. 2.

```
context InstanceSpecification
inv:
classifier->forAll(c | c.parents()->notEmpty() implies
  classifier->includesAll(c.parents()))
```

- `Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier.`
  This requirement focuses again on instances and not on classifiers. Instances may have slots for each structural feature of the classifier it instantiates. As generalization may be involved, also the structural features of the parent classifiers are accessible.

```
context InstanceSpecification
inv:
slot->forAll(s | classifier.attribute->includes(s.definingFeature.
                   oclAsType(Property)) or
                 classifier.parents()->exists(p | p.attribute->
                   includes(s.definingFeature.
                     oclAsType(Property))))
```

- `Any constraint applying to instances of the general classifier also apply to instances of the specific classifier.`
  Here, the constraint applies to *Classifier* and not its instances. The intention is that all constraints of classifiers apply also to their subclassifiers. Hence, the constraint must be linked - implicitly - to all subclassifiers.

```
context Classifier
inv:
parents()->notEmpty() implies
  parents()->forAll(p | self.constraint->includesAll(p.constraint))
```

Absent Static Semantics

As we have seen above, we can find a large amount of static semantics throughout the UML specification - some formalized, some not, some correctly associated to a modeling element, some not. Other constraints that are maybe too obvious are not mentioned at all, but should also be given in OCL to allow automated checks for well-formedness of models.

To give a simple example, each *NamedElement* can have a name, but must not as the multiplicity of the attribute *name* is 0..1. Each *Classifier* is a *Type* by inheritance just as *Class*, *Association*, or *DataType* who again inherit from *Classifier*. A *Type* should be identifiable by its name as the modeler must have a means to specify the type of a property. It is therefore reasonable to require that at least each *Class* and each *DataType* must have a name (a property cannot be typed by an *Association*):

```
context Class
inv:
  name->size() = 1

context DataType
inv:
  name->size() = 1
```

Another evident fact is that a class can only specialize another class while an association can only specialize another association. For *Class*, this fact is already mirrored in the abstract semantics, but not for *Association*. The same holds for *AssociationClass*. Hence, we require:

```
context Association
inv:
  general->forAll(c | c.oclIsTypeOf(Association))

context AssociationClass
inv:
  general->forAll(c | c.oclIsTypeOf(AssociationClass))
```

Other static semantics are left open intentionally due to the fact that UML can be used in various domains and in combination with various programming languages as target. One example is a semantic variation point for *Association*:

- `The interaction of association specialization with association end redefinition and subsetting is not defined.`

It is of course reasonable to define more precise constraints for modeling elements in specific applications of UML, e.g. in profile definitions as we can see in Sec. 4. It may be necessary to require that objects must have a name, or that redefining properties in generalizations must have the same name as the redefined one for code generation purposes. Nevertheless, such constraints should not be formalized in the UML metamodel in general as different projects have different requirements.

4 Well-formedness of Profiled Models by Example

Profiles as described in [17,18] are a mechanism to tailor the UML to specific application domains by (a) introducing new terminology, (b) introducing new syntax/notation, (c) introducing new constraints, (d) introducing new semantics, and (e) introducing further information like transformation rules. Changing the existing metamodel itself e.g. by introducing semantics contrary to the existing ones or removing elements is not allowed. Consequently, each model that uses profiles is a valid UML model. A UML2 profile consists mainly of stereotypes, i.e. extensions of already existing UML modeling elements. A UML modeling element is chosen as basis and add-ons are specified. In the following, we focus on the introduction of new static semantics with OCL.

Our example is the Railway Control Systems Domain (RCSD) profile [5,20] that is designed to be used for the development of railway controllers. Its static semantics

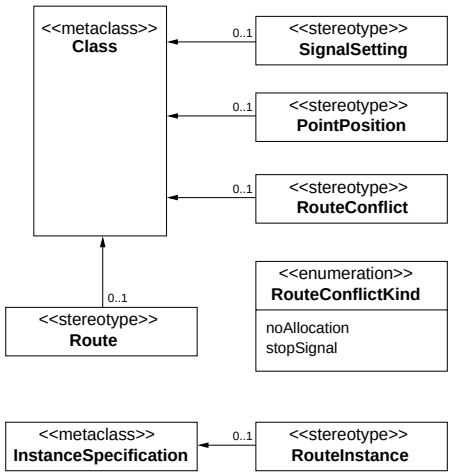

Fig. 3. Excerpt from RCSD profile

are completely defined with OCL [3,4]. Several stereotypes are defined that are elementary for the design of railway systems [19]: track segment, crossing, point, sensor, signal, and routes. In addition, domain-specific associations and datatypes are defined that are needed for accurate modeling of the domain. Base classes of the UML metamodel are *Class*, *Association*, and *InstanceSpecification*. An excerpt of the profile is shown in Fig. 3.

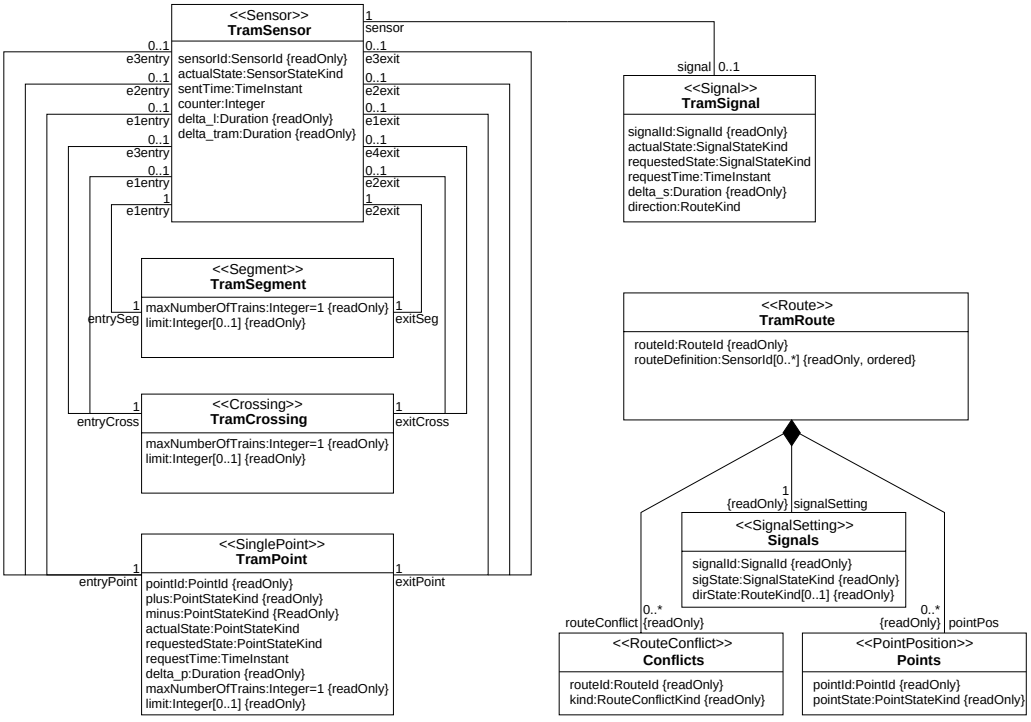

Fig. 4. Simple tram system - class diagram

The RCSD profile is intended to improve the collaboration between software developers and domain experts. A specific railway system, e.g. trams, railways in Germany, or railways in Great Britain, is modeled in class diagrams as we can see in Fig. 4. The concrete projection - that is a concrete track layout with routes - is than

modeled as an object diagram. Here, we provide also a domain-specific notation as shown in Fig. 5 to improve communication with domain experts. Class and object diagrams are automatically validated with respect to its static semantics as described in Sec. 5. Behavioral semantics are based on a state transition system that serves as foundation for code generation for controllers as well as formal verification by bounded model-checking. For details, we refer to [5].

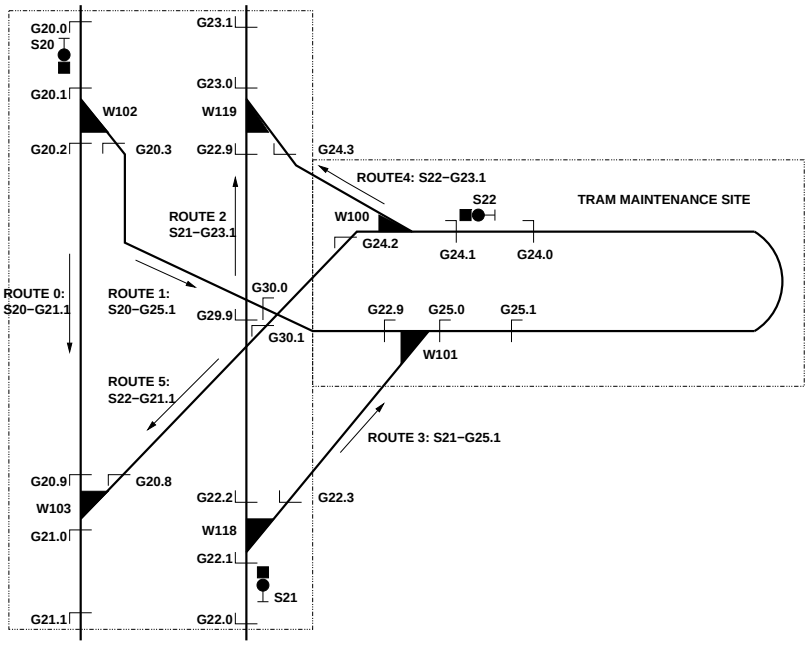

Fig. 5. Simple tram system - object diagram

OCL can be used in various ways to specify stereotypes precisely:

- Constraining property values: A stereotype has all properties of its base class and can add only attributes. Defining new associations to classes in the reference metamodel or other stereotypes is not allowed. Therefore, constraining values of existing attributes and associations is a useful means to give a stereotype the desired functionality.
- Specifying dependencies between values of different properties of one element: Often, it is necessary to describe dependencies between the values of properties of a modeling element precisely.
- Specifying dependencies between property values of different instances of one element: Some properties like identification numbers need specific values for different instances of one element.
- Specifying dependencies between property values of different instances of different elements: In the same way, several elements may have properties whose values have some kind of relationship. Here, it is important to chose the context of the constraint carefully such that the constraint is not unnecessarily complicated because another modeling element would have been the better choice as basis for the constraint.

Constraints on Classifiers

Each *Route* must have a constant attribute *routeId* with type *RouteId*:

```
context Route
inv:
ownedAttribute->one(a | a.name->includes('routeId') and
                        a.type.name->includes('RouteId') and
                        a.upperBound()->asSequence()->first()=1 and
                        a.lowerBound()->asSequence()->first()=1 and
                        a.isReadOnly = true)
```

To understand the structure of such a constraint, a look at the UML metamodel is helpful. As all network elements are stereotypes of *Class* from the UML2 *Kernel* package (see Fig. 1), we can refer to all properties of *Class* in our constraints. Properties on the model level are instances of class *Property* on the metamodel level, which are associated to *Class* by *ownedAttribute*. As a *StructuralFeature*, *Property* is also a *NamedElement*, a *TypedElement*, and a *MultiplicityElement*, which allows to restrain name, type, and multiplicity as shown in the constraints above. Such constraints are extensively used as RCSD diagrams must provide certain information for code generation purposes.

Constraints on Instances - Properties of One Instance

Another example are *Points* that are - together with *Segments* and *Crossings* - part of the track layout. Each *PointInstance* has a *plus* and *minus* position that is modeled as an attribute. One of these has to point *STRAIGHT* and the other one *LEFT* or *RIGHT*:

```
context PointInstance
inv:
slot->select(s1 | s1.definingFeature.name->includes('minus') or
                  s1.definingFeature.name->includes('plus'))->
  one(s2 | s2.value->size()= 1 and
           s2.value->first().oclIsTypeOf(InstanceValue) and
           s2.value->first().oclAsType(InstanceValue).instance.
             name->includes('STRAIGHT')) and
slot->select(s1 | s1.definingFeature.name->includes('minus') or
                  s1.definingFeature.name->includes('plus'))->
  one(s2 | s2.value->size()= 1 and
           s2.value->first().oclIsTypeOf(InstanceValue) and
           (s2.value->first().oclAsType(InstanceValue).instance.
              name->includes('LEFT') or
            s2.value->first()->oclAsType(InstanceValue).instance.
              name->includes('RIGHT')))
```

Constraints on Instances - Properties of Different Instances of One Classifier

To give an example for this constraint category, each *RouteInstance* requires a unique identifier. We must therefore assure that the set of all identifier values of all

route instances contains unique elements:

```
context RouteInstance
inv:
RouteInstance.allInstances->collect(slot)->asSet->flatten->
  select(s | s.definingFeature.name->includes('routeId'))->
    iterate(s:Slot;
            result:Set(LiteralRouteId) =
              oclEmpty(Set(LiteralRouteId)) |
            result->including(s.value->asSequence->first.
              oclAsType(LiteralRouteId)))->isUnique(value)
```

Constraints on Instances - Properties of Different Instances of Different Classifiers

Each *Route* is defined by an ordered sequence of sensors identifications. The signal setting for entering the route and sets of required point positions and of conflicts with other routes are further necessary information. This implies that the identification numbers belong to existing instances, e.g. the sensor identifications given in the definition of a route. Hence, the following constraint must hold for each *RouteInstance* with respect to the named *SensorInstances*:

```
context RouteInstance
inv:
let i:Set(Integer) =
  slot->select(s | s.definingFeature.name->includes
    ('routeDefinition'))->asSequence->first().value->
      iterate(v:ValueSpecification;
              result:Set(Integer)=oclEmpty(Set(Integer)) |
              result->including(v.oclAsType(LiteralSensorId).value))
in
  i->forAll(id | SensorInstance.allInstances->exists(sens |
    sens.slot->select(s | s.definingFeature.name->
      includes('sensorId'))->asSequence->first().value->first().
        oclAsType(LiteralSensorId).value = id))
```

5 Automated Validation with USE

The concrete validation process is performed with the tool USE [1] that expects a (meta)model in textual notation as input. For syntax, we refer to [22] and [23]. The tool implements the set-theoretic semantics described in [21] in detail. The key feature for our purposes is the interpreter that evaluates OCL constraints. The evaluation time is dependent on the number of elements in the model, the number of instances of the modeling element, the number of constraints and also the kind of constraints that range from very simple to quite complex as we have seen in Sec. 3 and Sec. 4. In general, we can only say that the complexity is polynomial with respect to the named inputs.

In our case, the input model is the UML metamodel - respectively a part of it that is necessary for class and object diagrams as described below. Further, profile

can be added to the metamodel. On this basis, instance models can be checked with respect to the invariants in the metamodel. The instance model consists of both class layer and object layer. A similar application of USE with respect to the four metamodeling layers of UML is shown in [13]. Here, the application of USE in metamodeling is shown by a small example.

Our metamodel is constructed from one input file for the UML2 metamodel and one input file for each applied profile. The result is one large metamodel. The reasons for this procedure is simple as we want to be able to check models that do not apply profiles and models that apply one or more profiles. Also, we are interested if a profile is compliant to UML. So far, no OCL tool is capable of checking the consistency of a set of OCL constraints. Therefore, we assume a profile compliant to UML as long as both the constraints in the metamodel and the constraints in the profile(s) are all valid which is of course no proof. At least, we are able to test the compliance of profiles and their reference metamodels by example.

Modeling the UML Metamodel and the RCSD Profile with USE

In the metamodel file, a description of classes with attributes and operations, associations, and OCL constraints is expected. OCL expressions are either invariants, definitions of operations, or pre-and postconditions of operations. Only operations whose return value is directly specified in OCL and not dependent on preconditions are considered side-effect free and may be used in invariants. For the validation process all invariants must be fulfilled by the instance model(s).

From the UML metamodel, the packages *Kernel*, *Dependencies*, *Interfaces*, and *BasicBehaviors* have been modeled with few changes: (a) The erroneous OCL statements have been corrected and additional constraints added as described in Sec. 3. (b) Some names - mostly association ends - had to be changed to guarantee unambiguous navigation in the model. (c) USE does not support *UnlimitedNatural* as type. This problem has been overcome by using *Integer* and additional constraints that restrict corresponding values to $\mathbb{N}$. We do not support nested packages. A short excerpt of the metamodel in the textual notation of USE is shown below:

```
class Association < Classifier, Relationship
attributes
  isDerived:Boolean
end
...
association Association2Type between
  Association[*]
  Type[1..*] role endType
end
...
context Association
inv Association_1:
  self.parents()->forAll(p |
    p.oclAsType(Association).memberEnd->size()=self.
    memberEnd->size())
```

Profiles are not directly supported by USE. This problem has been overcome by modeling each stereotype as a subclass from its base class, i.e. a metamodel extension. Modeling profiles as restricted extensions to metamodels is feasible with respect to [14]. Here, modifications to metamodels are classified in level one (all extensions to the reference metamodel allowed), level two (new constructs can be added to the referenced metamodel, but existing ones cannot be changed), level three (each new construct must have a parent in the reference metamodel), and level four (new relationships are only allowed as far as existing ones are specialized. The lower levels include all restrictions of the levels above. Therefore, profiles can be considered a level four metamodel extension and modeled as such in USE. [2] The profile designer must keep in mind that associations cannot be added and existing attributes and associations can only be restricted, e.g. my narrowing a multiplicity.

Checking Compliance on Class and Object Level

Evaluating constraints is possible for instances of the given (meta)model. In our case, this includes the class and object level of UML, as classes and instances are both defined in the metamodel and can be instantiated. Classes, associations, objects, links, etc. form one large instance model that is generated from a UML model created with some kind of CASE tool. The generation of USE code is decoupled from the CASE tool in use to preserve independence. Tool-specific is only the parser of the output of the tool. This step can be hopefully omitted in the future as we expect tools to comply to the XMI standard soon.

The generation of USE code itself is straightforward as the main tasks are creating instances of the elements in the input model, setting properties, and instantiating associations. The correct implementation of types, inheritance, and interfaces requires some attention, but is not complicated. Important is the correct order of instances creation as obviously the more general parts must be created before the specific parts of the instance model. The complexity - dependent on the number of classes, attributes, methods, objects, etc. never increases $O(n^3)$.

As an example, a tram network description is used on class level. Tram networks consist of segments, crossings, and single points that are all used unidirectionally. Furthermore, there are signals, sensors, and routes. This constellation is shown in Fig. 4. The class diagram is contained in one USE input file. An excerpt from the corresponding USE instance model on class level is shown in the following:

```
!create TramSensor:Sensor
!set TramSensor.name := Set{'TramSensor'}
...
!create sensorId:Property
!set sensorId.name := Set{'sensorId'}
...
!insert (sensorId, SensorId) into TypedElement2Type
!insert (TramSensor, sensorId) into Class2Property
```

A concrete network of a tram maintenance site with six routes is shown in Fig. 5.

[2] [14] considers profiles as level three which is incorrect as the relationship restriction has to be respected by profiles.

The explicit route definitions have been omitted for the sake of brevity, but can be easily extracted from the figure. This diagram has been used for the validation on the instance level. It consists of 12 segments, 3 crossings, 6 points, 25 sensors, 3 signals, and 6 routes, specified in a second USE input file. In this way, it is possible to create one input file for each object diagram and validate them separately in combination with the class diagram file. The object diagram files have the same appearance as the class diagram ones as we can see in the following excerpt:

```
!create s1:SegmentInstance
!insert (s1, TramSegment) into InstanceSpecification2Classifier
...
!create s1exit:Slot
!insert (s1exit, e2Exit) into Slot2StructuralFeature
!create s1exitValue:InstanceValue
!insert (s1exit, s1exitValue) into Slot2ValueSpecification
!insert (s1, s1exit) into InstanceSpecification2Slot
```

Results

In this example, all invariants have been fulfilled. The correctness of the OCL constraints can be easily checked by adding intentional errors like incorrect association ends or signals with the same identification number. USE facilitates tracing of such errors by (a) showing which instance of the metamodel has violated an invariant and by (b) decomposing the invariant in all sub-clauses and giving the respective evaluation.

Some effort has to be made with respect to the USE model. Once this model is ready, all UML models can be checked for well-formedness. The same holds for profiles; the UML metamodel must be extended for each profile to allow validation of models with USE. As this task is performed once per profile, the effort seems reasonable. With respect to the RCSD profile, the instance model on class level has to be modeled once per specific railway system, e.g. once for trams. With this part of the instance model, all kinds of concrete track layouts can be checked. The tram example consists of approximately 1500 lines of input data to USE dedicated to the class level. These are generated from class diagrams by parsing the output of CASE tools and mapping them to the USE input language. Concrete track layout are generated - also automatically - from object diagrams. In this way, all kinds of track layouts for one system can be checked. The example track layout requires about 5000 lines of USE code.

6 Discussion

The validation of UML models has been proven useful in several ways: (a) it can be shown that a model complies to the static semantics of UML, (b) it can be shown by example that a profile complies to the static semantics of UML, and (c) it can be shown that a profiled model is valid according to the - added - static semantics of the profile.

As the validation is performed automatically, it is highly useful to ensure the

soundness of a model before other tasks such as verification, simulation, or code generation are performed. The validation of the model for each of these tasks can be omitted as the model is assumed reliable. Another effect of the validation with USE is the improvement of the OCL constraints themselves. As most case tools have no OCL support, it is hard to detect if constraints exhibit syntax errors or if complicated constraints really have the intended meaning.

Some effort has to be made to attain this goal. First, static semantics of UML have to be improved as they are erroneous and incomplete. Second, static semantics for a profile must be defined formally. Both UML and a potential profile must be expressed as a USE model to allow automated validation. As these tasks are performed once (per profile), the effort seems reasonable. The generation of USE input code based on CASE tool output is performed automatically.

So far, most work on UML semantics focuses on behavioral semantics - e.g. [6], [7], [9], or [12] - while static semantics are not discussed in detail. As we can see from these examples, these behavioral semantics differ from approach to approach; not at least a result from the different application domains their developers had in mind. This seems appropriate as each domain has it own needs. In contrast, we believe that static semantics for UML can be unified and formalized as shown in this paper to allow automatic validation. The resulting models are reliable and can be used in further tasks, independent from the application domain.

The validation of OCL constraints of the UML metamodel with USE has been performed for UML1 and UML2.0 before as shown in [22] and [2]. This has been extended to UML2.1 in this work. We are not aware that a thorough correction of existing constraints and formalization of constraints in natural language and of absent constraints has been performed before for class and object diagrams of UML2. In [11], similar work is done with respect to the static and behavioral semantics of statecharts given in natural language in UML2. The usage of OCL in metamodeling has been suggested in [13], but the application of this concept for profiles and the automatic validation of profiled models designed with CASE tools has not been performed before. More details with respect to the example profile can be found in [4].

The validation of profiles has been shown by an example. An adaption of the validation process to other profiles can be performed straightforward as the same kinds of constraints should appear. Validation is sensible for each profile whose application relies on a solid and unambiguous model. The RCSD profile used as example has shown that this is possible for real-world applications and problems. Even complex constraints can be formalized with OCL. Future work should investigate the usage of OCL for the formalization of static semantics of other profiles.

Also, the improvement of static semantics of UML should be pushed further to stabilize the backbone of the language. In this paper, only the basic features of UML in class and object diagrams have been considered. More work in this direction seems reasonable as a new UML standard cannot be expected in the next few years as we know from experience from the standardization of UML2. Furthermore, results for static semantics are valuable as they can be incorporated in new UML versions.

References

[1] *A UML-based Specification Environment*, http://www.db.informatik.uni-bremen.de/projects/use/.

[2] Bauerdick, H., M. Gogolla and F. Gutsche, *Detecting OCL Traps in the UML 2.0 Superstructure*, in: T. Baar, A. Strohmeier, A. Moreira and S. J. Mellor, editors, *Proceedings 7th International Conference Unified Modeling Language (UML'2004)*, LNCS **3273** (2004), pp. 188–197.

[3] Berkenkötter, K., *Design of a Railway Domain Profile and its OCL-based Validation*, in: B. Demuth, D. Chiorean, M. Gogolla and J. Warmer, editors, *OCL for(Meta-)Models in Multiple Application Domains 2006*, Electronic Communications of the EASST **5** (2007).

[4] Berkenkötter, K., *OCL-basesd Validation of a Railway Domain Profile*, in: T. Kühne, editor, *Models in Software Engineering*, LNCS **4364** (2007), pp. 159–168.

[5] Berkenkötter, K. and U. Hannemann, *Modeling the Railway Control Domain rigorously with a UML 2.0 Profile*, in: J. Górski, editor, *Safecomp 2006*, LNCS **4166** (2006), pp. 398–411.

[6] Breu, R., U. Hinkel, C. H. andCornel Klein, B. Paech, B. Rumpe and V. Thurner, *Towards a Formalization of the Unified Modeling Language*, in: M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97 - Object-Oriented Programming*, LNCS **1241** (1997), pp. 344–366.

[7] Clark, T. and A. Evans, *Foundations of the Unified Modeling Language*, in: *2nd Northern Formal Methods Workshop*, electronic Workshops in Computing (1998).

[8] Clark, T., A. Evans, P. Sammut and J. Willans, "Applied Metamodeling," Xactium, www.xactium.com, 2005.

[9] Damm, W., B. Josko, A. Votintseva and A. Pnuelli, *A Formal Semantics for a UML Kernel Language*, http://www-omega.imag.fr/doc/d1000009_6/D112_KL.pdf (2003).

[10] Evans, A. and S. Kent, *Core Meta-Modelling Semantics of UML: The pUML Approach*, in: *UML*, LNCS **1723** (1999), pp. 140–155.

[11] Fecher, H., J. Schönborn, M. Kyas and W. P. de Roever, *29 New Unclarities in the Semantics of UML 2.0 State Machines*, in: K.-K. Lau and R. Banach, editors, *ICFEM*, LNCS **3785** (2005), pp. 52–65.

[12] Fischer, C., E.-R. Olderog and H. Wehrheim, *A CSP View on UML-RT Structure Diagrams*, in: H. Hußmann, editor, *FASE*, LNCS **2029** (2001), pp. 91–108.

[13] Gogolla, M., J.-M. Favre and F. Büttner, *On Squeezing M0, M1, M2, and M3 into a Single Object Diagram*, Technical Report LGL-REPORT-2005-001, Ecole Polytechnique Fédérale de Lausanne (2005).

[14] Jiang, Y., W. Shao, L. Zhang, Z. Ma, X. Meng and H. Ma, *On the Classification of UML's Meta Model Extension Mechanism*, in: *The Unified Modelling Language: Modelling Languages and Applications*, 2004, pp. 54–68.

[15] Object Management Group, *MDA Guide Version 1.0.1*, http://www.omg.org/docs/omg/03-06-01.pdf (2003).

[16] Object Management Group, *Object Constraint Language*, http://www.omg.org/docs/formal/06-05-01.pdf (2006).

[17] Object Management Group, *Unified Modeling Language: Infrastructure*, http://www.omg.org/docs/formal/07-02-06.pdf (2007).

[18] Object Management Group, *Unified Modeling Language: Superstructure*, http://www.omg.org/docs/formal/07-02-05.pdf (2007).

[19] Pachl, J., "Railway Operation and Control," VTD Rail Publishing, Mountlake Terrace (USA), 2002, iSBN 0-9719915-1-0.

[20] Peleska, J., K. Berkenkötter, R. Drechsler, D. Große, U. Hannemann, A. E. Haxthausen and S. Kinder, *Domain-Specific Formalisms and Model-Driven Development for Railway Control Systems*, in: *TRain workshop at SEFM2005*, 2005.

[21] Richters, M., "A Precise Approach to Validating UML Models and OCL Constraints," BISS Monographs **14**, Logos Verlag, Berlin, 2002, Ph.D. thesis, Universität Bremen.

[22] Richters, M. and M. Gogolla, *Validating UML Models and OCL Constraints*, in: A. Evans, S. Kent and B. Selic, editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, LNCS **1939** (2000), pp. 265–277.

[23] Richters, M. and M. Gogolla, *OCL: Syntax, Semantics, and Tools*, in: T. Clark and J. Warmer, editors, *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, LNCS **2263** (2002), pp. 42–68.

The HiVe Writer

Tony Cant[1,2], Ben Long, Jim McCarthy, Brendan Mahony, Kylie Williams

Command, Control, Communications and Intelligence Division
Defence Science and Technology Organisation
PO Box 1500, Edinburgh, South Australia 5111

Abstract

Critical systems require assurance that key security, safety or mission requirements are met. Tools are necessary to provide this assurance. The HiVe Writer supports model-based documentation for complex critical systems. The HiVe Writer forms the functional foundation for the ambitious HiVe (Hierarchical Verification Environment) project which aims to provide a unified framework in which entire design projects can be described with the highest level of assurance. The primary innovation in the HiVe Writer is a centrally-managed design model: any design, explanatory and technical documents created within the tool are constrained to be consistent with this design model and therefore with each other. This paper gives a detailed description of the HiVe Writer, showing how it supports model-based editing of structured technical documents and, in particular, requirements formulation.

Keywords: Critical Systems, Formal Methods, Design Verification, Requirements Formulation.

1 Introduction

The Australian Government Department of Defence procures a number of computer-based systems that are *critical* in the sense that they have requirements whose violation could have grave consequences.

- *security-critical systems* need to satisfy a *security policy* [4] enforcing a number of security requirements, such as the prevention of unauthorised access to confidential data or the maintenance of operational integrity.
- *safety-critical systems* need to satisfy a *safety case* [7] ensuring that system design eliminates or reduces the chance that hazardous system states (that could lead to injury or death) are reached.
- *mission-critical systems* must ensure that certain goals, mission or performance requirements (critical to success of the mission) are met.

[1] The authors wish to thank the Defence Materiel Organisation for sponsorship and funding of The HiVe.

[2] Email: Tony.Cant@dsto.defence.gov.au

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The central challenge for critical systems is to be able both to achieve assurance and to transfer this assurance to other parties. Assurance is achieved by carrying out a range of specification and verification activities in the early stages of development that show how critical system requirements are met. The transfer of assurance involves an appropriate mix of informal, semiformal and formal arguments that will convince a third party, such as an evaluator or certifier.

An example of a tool supporting informal arguments is Telelogic's DOORS [2]. DOORS provides a framework in which users can generate structured documentation consisting of a series of numbered informal requirements.

Semi-formal arguments are supported by tools such as Telelogic Rhapsody, which combines requirements management and design and simulation of UML [10] schematics within the one tool. The MathWorks Simulink [8] is another example of a semi-formal development tool and supports the design and simulation of functional block diagrams.

Formal methods provide well-defined languages such as Z [6] and CSP [5] that allow precise specifications to be written and subsequent rigorous verification procedures — typically *model checking* [3] or *theorem proving* [11,9] — to be performed.

Regardless of whether informal, semiformal or formal arguments are used, or whether a single formalism or multiple formalisms are used, the need for assurance requires documentation that is well structured and that provides consistency between all arguments. The authors' contention, based on extensive research and application, is that the following are the key desiderata for a design tool to support the achievement and transfer of assurance for critical systems:

Powerful Modelling: The tool should support the design of hierarchical, concurrent and real-time systems.

Trustworthy Modelling: The tool should support reasoning and proof as well as simulation.

High-Assurance: The tool should support the specification and verification of critical requirements, as well as the ability to flow requirements to system components.

Communication: The tool should be document-driven, supporting different views for different audiences.

Synergy with other tools: The tool should allow the interaction with other tools, such as theorem provers, model checkers and simulation tools.

The *Hierarchical Verification Environment* (HIVE) project aims to realise these goals by providing a unified framework in which entire design projects can be captured. It will encourage designers to write design, explanatory and technical documents in parallel, thereby helping the user to produce output that will convince others of the correctness of the design. All documents will be created (and remain) consistent with a centrally-managed document-driven design model, which will allow consistent presentation of informal, semi-formal, and formal arguments, and from which various tools involving simulation, model checking and theorem proving can be invoked. The HIVE *Writer*, discussed in this paper, is the component specifically concerned with document generation for achieving this.

In Section 2 we introduce the fundamental principles on which the Writer is based, and give an intuitive description of the mechanisms through which it realises the goals outlined above. Section 3 sketches the functional decomposition of the Writer's implementation. In

Section 4, we consider the use of the Writer, from a user interface viewpoint, in a case study based on the development of the Def (Aust) 5679 Safety Standard [1]. Section 5 presents some conclusions.

2 The Writer: a mechanistic underview

The fundamental principles underlying the HIVE Writer are embodied in the dual notions of *model-based editing* and *literate modelling*:

(i) The Writer builds a *model* as an approach to complex documentation tasks. This helps the user to maintain consistency between documents since they all refer to the same underlying model. For the same reason it helps the user to oversee and implement change management. Finally, the model itself provides assurance of consistency in the design process.

(ii) The Writer builds *literate documentation* as an approach to complex modelling tasks. This helps to promote the user's comprehension of the design and implementation. Equally importantly, it helps to communicate the requirements, the design, and the results of assurance activities to other stakeholders in the process.

The HIVE Writer can thus be regarded as an application of *lightweight formal methods*: the tool enforces consistency as the model-based documentation is built, but the user is not required to be an expert in the application of formal methods.

Our approach to implementing a tool that upholds the above principles utilises highly structured documents that are correlated through a central fact repository. In this section we briefly elaborate on how this works.

2.1 Structure algebraically

The document structure is induced from the use of what we call *structured text*. At the simplest level, structured text is the language generated by a repository of *syntax constructors*. The constructors are *sorted*: i.e., each constructor argument only accepts terms from a particular subset of structured text (labelled by a so-called *sort*), and the constructor returns a term of definite sort. Documents created in the HIVE Writer are written in structured text, thereby acquiring a hierarchical structure from the sort-directed nesting of constructors.

Figure 1 illustrates the hierarchy exhibited by the mathematical expression

$$x + y = 3 \wedge 2x - y = 3\,.$$

The structure, in this simple example of a term algebra, branches according to the algebraic nature of the operators: the top level is a conjunction of two subexpressions; each subexpression is an equality; and so on. For example, the top-level conjunction is achieved through the constructor $\wedge$, for which a production is indicated via a *dummy box* inscribed with the required sort.

The syntax constructors need not just build mathematical terms, but could act on formatted text or whatever is desired. One example, featuring in Section 4, is a simple requirements language. It is used to enforce a strict documentation standard on requirements gathering which prevents the user from writing requirements on development artifacts that cannot be held to account.

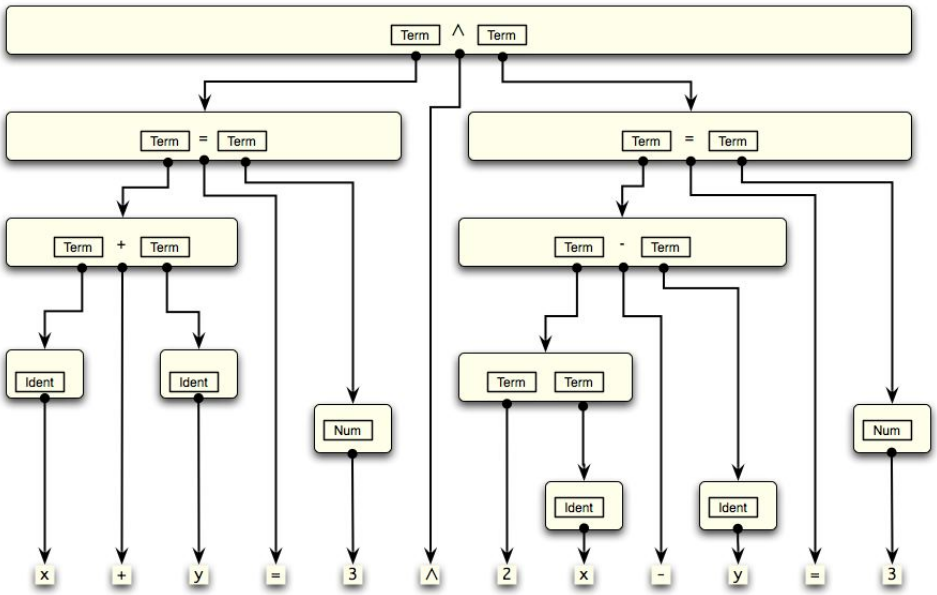

Fig. 1. Structured text.

To remove any prejudice of usage, we introduce the term *element* to denote the fundamental data underlying a "syntax constructor": i.e., a *name*, a *result* sort and the list of argument sorts known as its *arity*. For example, the top-level element in Figure 1 could be denoted

$$\texttt{conjunction} :: \text{Term} \leftarrow [\text{Term},\text{Term}]$$

In the HIVE Writer, elements are used at all levels in document production. Technically, they provide the constructors of a *many-sorted algebra*. Elements must be defined for *all* language elements required to build documents; in particular, those required to declare other elements. So, a sort is an element, as are all the constructors required to build the sort list for the element arity.

2.2 Present for readability

Truly literate documentation must be attractive for readability. Structured text would not be suitable for producing literate documents if it did not allow the syntactic freedom to display documents in their most natural and informative style as appropriate to a given audience. Indeed, in Figure 1 the raw element data does not appear – rather, a corresponding presentation using appropriate mathematical symbols is given. For example, the element `conjunction` is presented by $\square \wedge \square$.

The HIVE Writer requires the user to define a *production* corresponding to each element, and it is provided at the point of declaration of the given element. The production has the form of a sequence of the element's arguments sandwiched by delimiters. The delimiters of the production can be arbitrary Unicode modified by the standard text attributes (bold, italic, color, etc). This provides sufficient freedom for most application domains, and is particularly straightforward for rendering to the screen. Thus, the document as presented on the screen is highly readable, and can be faithful to the printed form.

In fact, the user can define more than one production for a given element if desired.

The different productions are distinguished by assignment to separate *presentation styles*. The mandatory production required at element declaration is assigned to the default style. An ordering on the different styles can be introduced, in which the default style provides a top. If a given element does not have a production in a given style, the style ordering is followed until a production is found. The ability to change presentation at a global level can be very useful: e.g., the documents can be "toggled" between presenting mathematical expressions as structured English statements or algebraic terms by the choice of appropriate presentation style.

We introduce the term *presentation* to denote the fundamental data required to present a given element: i.e., for a given element name, the production and the style to which it is assigned.

2.3 Reference for consistency

The consistent correlation between documents is built up through the use of *references* to a central fact repository. Thus, the repository stores a table of elements and a table of presentations. In use — such as in the example of Figure 1 — the elements are instantiated *by reference* to the repository. In a document, a given element reference includes the name of the desired element and the desired presentation style. The document is then rendered to the screen with the element reference replaced by the corresponding production of that style.

Element references provide the skeleton of structured text as described in Section 2.1 above. They are actually an instruction for constructing the syntax constructor corresponding to a given element using data from various tables in the repository. We will also extend the structured text to incorporate *direct* reference to entries in repository tables. We mentioned just two such tables in the previous paragraph, but in fact many tables are required to support the basic Writer functionality – and there will be many more as the HIVE is developed. The different tables store different classes of *facts* about elements. Note, for example, that a presentation can be considered to be a fact about a given element: the fact that in the given style the element will be presented by the given production. The individual chunks of data – the style, or the production – will be referred to as fact *attributes*. Thus, these direct references will be referred to as *attribute references*.

We have already introduced facts about elements other than presentations. There is a table that records the ordering on styles. We also have a table that records a similar ordering on sorts – an ordering that introduces subsorting to the language.

2.4 Structured text in model-based documentation

The overall user goal in the HIVE is to create a *project*: a collection of related analysis constructs and documentation for a given activity. The concept of project encompasses a wide span of activities: from writing a technical paper through to a major system design and beyond. Documents required in a project will often be prescibed for a given activity, but may be extended to include any view of the project – say, pedagogical or summary form, pictorial or slideshow – that the user desires to construct.

The user does not start each project from scratch: a given project is built as an extension of existing projects, which are said to be *included*. In particular, the user will have a collection of analysis tools available, each of which can be controlled through the HIVE

Writer. That is, each tool will have a plug-in that defines an interface to the Writer and a corresponding collection of elements, facts and tables needed to model the interactions with the tool. We call this combination a HIVE *module*. The developers of a HIVE module must construct the base element algebra for recording tool interactions and outputs. Modules may also add additional User Interface structures – such as buttons and menu items – to allow tool-specific commands to be run from within the HIVE. At the bottom of the project hierarchy is the core Writer "module" that includes the repository structure discussed above and certain native tool mechanisms discussed below.

The HIVE inextricably mixes domain modelling with document construction. The user selects the HIVE modules (that is a standard *vocabulary*) relevant for the given activity and includes the corresponding projects in the current project. This give them a powerful, but highly structured framework for expressing their ideas. The user then introduces the specific elements and facts required to develop a model of the activity at hand to the desired level of detail. The grammar of the document language is derived from the total collection of elements. During final document preparation, the repository provides easy access to all elements and all facts about those elements defined in the current project as well as those in included projects. This adds to the technical quality of the product by enforcing consistency in vocabulary and facts. The user is able to tailor the presentation of the various elements (and hence facts) to different audiences by defining appropriate productions, thus adding to communicative effectiveness of the documents produced.

We can now summarise the different ingredients of what is meant by the rubric "structured text".

- Element references that provide the skeleton of structured text.
- Attribute references that allow re-use of pre-defined fragments of structured text.
- The dummy box that indicates a placeholder for structured text entry.
- Literals that allow "keyboard" entry of strings, numbers, *et cetera*.

The on-screen (and on-paper) presentation of structured text is controlled through the setting of standard text attributes such as type face, weight, shape, but also through an additional text attribute called presentation style that controls the actual characters used to represent the various elements. Text attributes may be inherited through the structured text hierarchy or set locally.

2.5 Supported mechanisms

The HIVE Writer provides native support for building tables. A functional view is given in the next section. Here we just point out two mechanisms which derive from the constructs developed above.

(i) **Syntax-direction:** The sort-direction explicit in the element algebra can be directly harnessed for syntax-directed editing capabilities. Observe first how Figure 1 can be read "downwards" as a temporal record of the construction of the mathematical expression. At each stage the user selects a dummy box and inserts either an element reference (with further dummy boxes for any arguments), or alphanumeric data from the keyboard. By including the dummy box as part of structured text we ensure that each stage in the construction is a legal expression.

The Writer provides a palette of syntax constructors for data-entry of element references, and a view of the tables that can be used to enter attribute references. Both windows provide syntax-direction by reflecting the allowed sort at the insertion point: data with the wrong sort are "greyed-out" and cannot be selected.

(ii) **Intelligent change management:** Just one important consequence of the referencing structure is to provide a straightforward facility for intelligent propagation of changes throughout the documents of a given project. The change is made once at a central point – wherever the data is entered into the repository. Since all other instances throughout the documentation are references to facts in that repository, the change is immediately propagated. We provide a simple illustration of this point in Section4.

3 The Writer: an application overview

As discussed, the HIVE Writer supports the preparation of structured technical documents. Thereto, it has three principal components.

(i) The *Document* editor which is for producing and updating structured documents.

(ii) The *Datastore* which is the combination of fact repository and syntax-directed data-entry mechanism discussed earlier. It provides the central design model for the project. All documents created within the project will be consistent with this design model and therefore with each other.

(iii) The *Tool Interface* which is an interface, with a flexible plug-in architecture, between the Datastore and the tools used to process the design constructs.

The user controls tool interactions through a distinguished document, called the *Normative Design Document* (NDD). The NDD is strongly coupled to the Datastore: it provides the declaration point for all entries in the Datastore from the current project. There is exactly one Datastore, and so exactly one NDD, in any given project.

3.1 Tool control in the NDD

The NDD is a structured document in which the user records the epistemic narrative of the project development. It is special, however, as we will now explain: it is a (literately programmed) script through which the user controls all interactions with tools.

The HIVE can potentially interface to a broad range of external tools, such as: theorem provers and model checkers; algebraic analysis and numerical analysis; system simulation; programming support; drawing; project management; requirements analysis; and version control. Moreover, the population of the Datastore tables is considered an interaction with the core Writer tool. The only assumption of the HIVE is that a supported tool must apply some function to given input data, and produce diagnostics and output data.

The uniformity of the tool interaction is supported by the concept of *commands*. Commands are collected in the DataStore as a special sort of element in the algebra of structured text. The commands required for interaction with a particular tool are defined in the module that supports that tool along with the code necessary to enact the interaction.

Each tool interaction proceeds as follows.

The syntax constructor for the command is inserted in the NDD, and the input arguments filled in, using standard Datastore/keyboard data-entry mechanisms. The input argu-

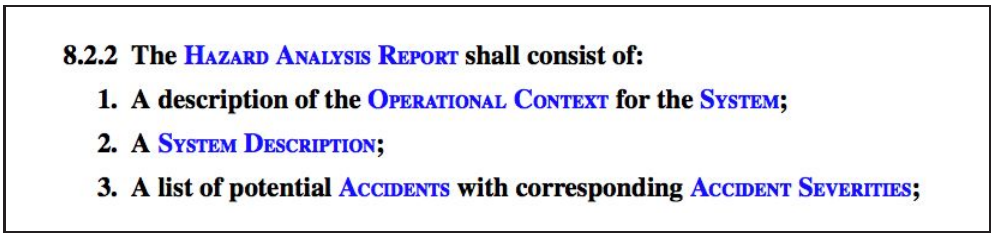

Fig. 2. A snapshot of Def (Aust) 5679.

ments encode the user input required for a given tool action. Some arguments to a command may be reserved for presenting the results of a tool interaction and the user is prevented from editing them. *Every* tool command (such as a proof step in a theorem prover tool) is constructed similarly.

The user then instructs the HIVE to *process* the command. Control is passed to the relevant module which then enacts the desired transaction with the tool. Results from the tool transaction may be presented in any output arguments of the command and one or more facts are entered into the Datastore. Error messages are handled and displayed appropriately to the user.

If the user is unhappy with the results, the command may be unprocessed to allow editing of command input and then re-processed. The updated results are then entered into the Datastore.

The NDD is a *linear* script, and thus processing a region of commands is simply sequencing the intermediate steps. Linearity allows the user to keep contextual focus whilst controlling several tools. Moreover, the script *must* be constructed in a well ordered fashion as determined by the execution logic of the tools it is controlling: for example, a theorem prover tool will not accept constants that have not been declared at an earlier stage.

Since commands are easily recognised in structured text, the command script can be embedded in an arbitrary pedagogical development of structured text. Ultimately, the universality of the command concept leads to the tight correlation between the Datastore tables and the NDD: modulo the pedagogical text, the data of the processed NDD command script is precisely equivalent to the current project's Datastore entries. Thus, indeed, the NDD is a literate exposition of the entire project construction.

4 Example: The Def (Aust) 5679 Safety Standard

To illustrate the HIVE Writer in use we consider the recent revision of the Australian Defence Standard Def (Aust) 5679 [7]. This Standard provides detailed requirements and guidance on the structure of the *Safety Case* for a safety-critical system. It focuses on *assurance* activities: these are system development and analysis activities that provide evidence that the system meets its safety requirements.

Our example is based on Chapter 8 (*Hazard Analysis*). Hazard Analysis is the first phase of Safety Case Development. Paragraph 8.2.2, shown before revision in Figure 2, highlights a frequent problem in the drafting of requirement documents: that of ensuring the assignment of clear lines of responsibility while maximising readability. The requirement, as stated, places an obligation on a document, the Hazard Analysis Report. This is hard to enforce. On the other hand, clarifying the responsible agent (in this case the Supplier) *in situ* would result in an overly complex statement of the requirement.

8.2.2 The SUPPLIER is responsible for compiling the HAZARD ANALYSIS REPORT.

8.2.3 The HAZARD ANALYSIS REPORT must include:

1. **A description of the OPERATIONAL CONTEXT for the SYSTEM;**
2. **A SYSTEM DESCRIPTION;**
3. **A list of potential ACCIDENTS with corresponding ACCIDENT SEVERITIES;**

Fig. 3. The revised Safety Standard document

The solution adopted by the authors of Def (Aust) 5679 is a two stage assignment of responsibility: firstly, responsibility for a document (or a process) is assigned to an appropriate agent; then requirements are placed on the document itself. This allows clear assignment of responsibility while ensuring the requirements are stated in a natural way. The resulting revision is shown in Figure 3.

The following describes a simple HIVE module that we have developed to support this discipline by enforcing the use of a constrained requirements language.

4.1 The Requirements Module

The Requirements module consists of a plug-in that provides special commands for constructing requirements and guidance paragraphs. The requirements command enforces a discipline so that

- each paragraph constructed in this manner is labelled uniquely; and
- the requirement text can only be formed with provided syntax constructors.

The latter is achieved through the Writer's sort-driven editing via a model for the requirements gathering process based on Figure 4. The boxes hold the sorts that distinguish the entities of the process, whilst the diamonds hold the syntax constructors that express the allowed relationships between entities.

As we will elaborate below, the resultant language cannot express the requirement in Figure 2, but *can* express the two-stage assignment of Figure 3. The plug-in also provides more subtle support. The user is protected from badly implementing the two stages: i.e., if the plug-in processes a 'must' requirement for a given Target then it also checks that a corresponding responsible Agent has been assigned – via an 'is responsible for' requirement

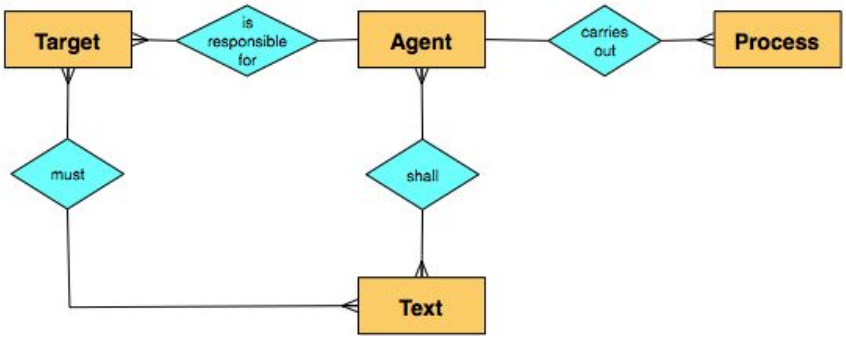

Fig. 4. Entity Relationship diagram of the requirements model.

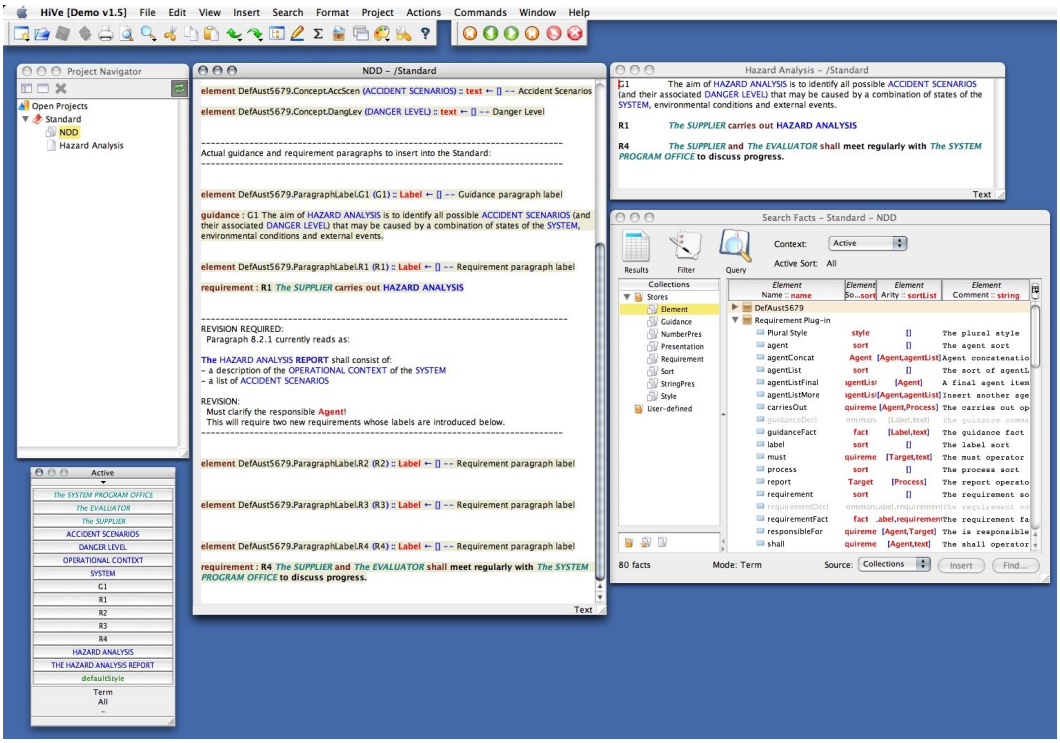

Fig. 5. A view of the HIVE in action

– before allowing the document to be processed.

4.2 Using the HIVE *on Def (Aust) 5679*

A typical session for the Def (Aust) 5679 project in the HIVE Writer is shown in Figure 5. The top left hand window is the Project Navigator which provides an easy mechanism for moving between the different documents in different open projects. The middle window is the NDD. It begins by instantiating the particular Agents (*Acquirer*, *Supplier*, *Auditor* and *Evaluator*), Processes (*Hazard Analysis*) and Targets (*Hazard Analysis Report*) used by Def (Aust) 5679 Chapter 8. It also contains some informal text flagging the need for revision of the badly-worded requirement. The top right hand window shows the Hazard Analysis document at the current stage of revision.

The Search Facts window, bottom right, is a view on the Datastore currently opened to show the Requirement plug-in elements. Finally, at the bottom left, is a palette derived by filtering a different Datastore view. It contains all the syntax constructors defined to date Def (Aust) 5679 in the project. In each project session any number of independent Datastore views (and syntax palettes) can be open, allowing multiple views into different aspects of the project data.

4.3 Revising the requirement 8.2.2

The requirement 8.2.2 must be replaced by a two-stage assignment as discussed earlier. For, suppose we were to attempt to introduce the previous requirement. Thereto, we insert the requirement command, provide a label (R2) for this command, and insert the ‘shall’ constructor to arrive at the situation shown in Figure 6. However, as seen there, an *Agent*

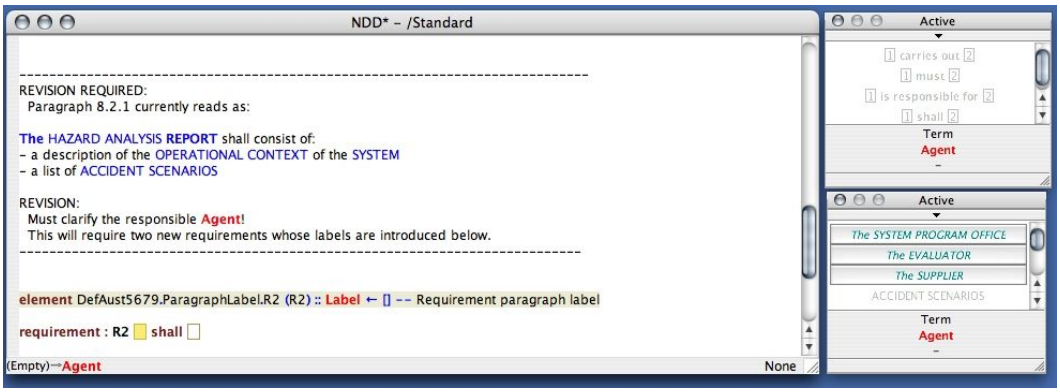

Fig. 6. The 'shall' constructor requiring an element of sort **Agent**

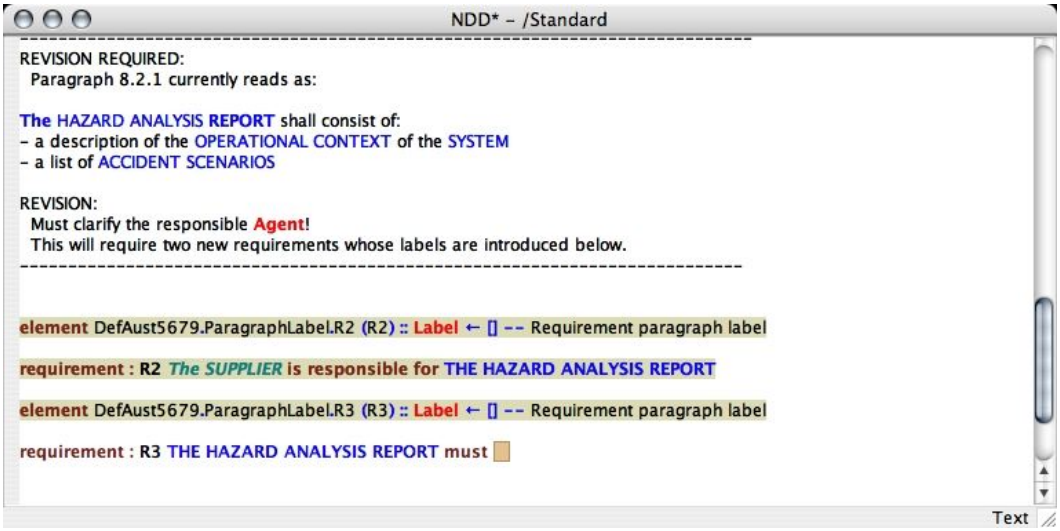

Fig. 7. Completing the text of the final requirement in the Standard.

must be supplied in the first argument (currently selected) of the constructor:

- the status bar (bottom left) displays the sort required to fill the selected dummy box;
- the Writer has 'greyed-out' in the palettes all syntax constructors that do not result in the Agent sort.

This feedback allows the user to be contextually aware at all stages in the construction.

The proper application of two-stage assignment needs *two* requirements to replace the current one, as seen in Figure 7. The first requirement establishes the responsible Agent using the 'is responsible for' constructor – we choose The Supplier as the responsible Agent and The Hazard Analysis Report as the corresponding Target. The second requirement establishes the contents of The Hazard Analysis Report using the 'must' constructor. Figure 7 shows the NDD waiting for the text stating the contents. On processing, the requirements are added to the DataStore, as shown in Figure 8. Finally, these requirements are added by reference the Safety Standard document leaving the result shown in Figure 3.

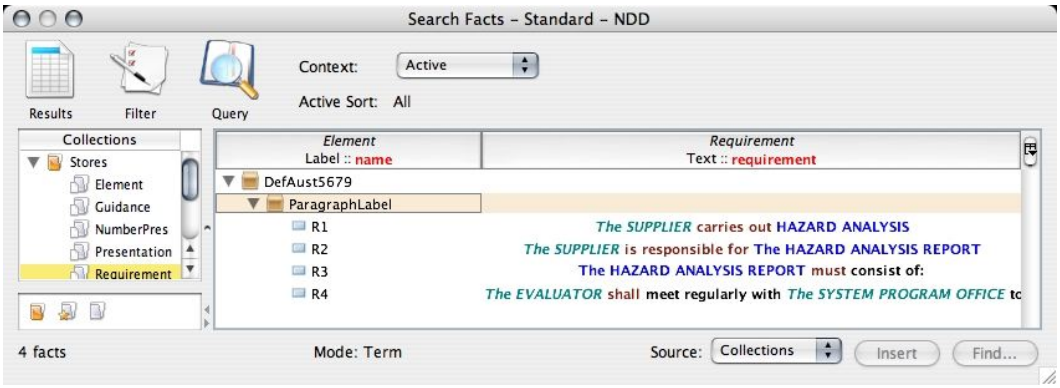

Fig. 8. The two-stage assignment added to the Datastore.

5 Conclusion

In this paper, we have given an overview of The HiVe Writer and shown how it can be applied in the case of a very simple example: a Safety Standard. We have shown how the production of the Safety Standard was supported through the construction of a light-weight model of the system being described. This model consisted of elements such as the Supplier and the Hazard Analysis Report as well as a structured language for describing the requirements placed on them by the Safety Standard (see Figure 4) The use of this model, along with the syntax-directed editing capability of The HiVe, ensures that users are constrained to write only paragraphs that conform with the model.

The Writer is still under development. Informing its development are a number of security devices and safety-critical systems. These examples are of great value in determining the best mode of user interaction with The Writer.

Future work will build upon the Writer: first of all, the Prover will incorporate the Isabelle theorem prover as a plug-in, extending the Writer by adding the ability to carry out literate theorem proving in Higher-Order Logic (a powerful and widely used framework for formal reasoning). The Modeller will enhance the Writer's functionality to allow for fragments of design (i.e. dataflow diagrams and state machines) to be stored in the datastore and referenced in documents. The Modeller will thereby exploit the Prover to support system modelling and verification.

References

[1] Australian Government Department of Defence. *Def (Aust) 5679, Issue 2: Safety Engineering for Defence Systems*, to appear in 2007.

[2] Tony Cant, Jim McCarthy, and Robyn Stanley. Tools for requirements management: a comparison of telelogic doors and the HiVE. General Document DSTO–GD–2006–0466, Defence Science and Technology Organistion, 2006.

[3] E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001.

[4] Common Criteria Project Sponsoring Organizations. *Common Criteria for Information Technology Security Evaluation*, version 2.1 edition, 1999.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[6] ISO/IEC 13568. *Information techonolgy – Z formal specification notation – Syntax, type system and semantics*, first edition, July 2002.

[7] Land Engineering Agency, Australian Government Department of Defence. *Def (Aust) 5679: Procurement of Computer-Based Safety Critical Systems*, 1998.

[8] The MathWorks. *Simulink 6 Reference*, 2007. http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/slref.pdf.

[9] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] Object Management Group. *UML Superstructure, OMG document formal/07-02-05*, v2.1.1 edition, 2007. http://www.omg.org/cgi-bin/doc?formal/07-02-05.

[11] J. M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, Germany, 2001.