Advancing Deductive Program-Level Verification for Real-World Application

Lessons Learned from an Industrial Case Study

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation

von
Thorsten Bormer
aus Koblenz

Datum der mündlichen Prüfung:	23. Oktober 2014
Referent:	Prof. Dr. Bernhard Beckert
Korreferenten:	Dr. Claude Marché
	Assoc. Prof. Dr. Wolfgang Ahrendt

Contents

List of Figures and Tables

List of Algorithms and Listings

Acknowledgments

This work would not have been possible without the help of many people – I am deeply grateful for all who have supported me in one way or another.

It is a great pleasure to thank my adviser Prof. Dr. Bernhard Beckert for giving me the opportunity to work on rewarding and challenging research topics, for the many insightful discussions, his advice and for encouraging me to pursue my research activities. Without him, this thesis would not have been written.

Among all my colleagues at the University of Koblenz-Landau and the Karlsruhe Institute of Technology I have had the pleasure to work with, I would like to thank in particular Dr. Vladimir Klebanov for always giving me critical and valuable feedback over the years since my time as student assistant in Koblenz, and Prof. Dr. Gerd Beuster for guiding my first steps in the Verisoft project. I also would like to thank Prof. Dr. Ulrich Furbach, as well as Prof. Dr. Peter H. Schmitt and my colleagues for fruitful discussions, helpful comments and for creating a stimulating and enjoyable working environment.

I would like to express my gratitude towards all those who contributed towards the completion of this thesis, either through great collaborations in research projects, in writing publications or teaching, in particular: Dr. Christoph Baumann, Holger Blasum, Daniel Bruns, Dr. Christoph Gladisch, Simon Greiner, Tianhai Liu, Florian Merz, Dr. Carsten Sinz, Dr. Mattias Ulbrich, Dr. Sergey Tverdyshev, and Dr. Markus Wagner. I am also greatly indebted to the members of the VCC development team and Hypervisor verification project within Verisoft XT for their assistance with the usage of the VCC verification tool and specification methodology.

I would also like to thank Dr. Claude Marché and Assoc. Prof. Dr. Wolfgang Ahrendt for accepting to be my co-examiners and for the valuable feedback.

A very special acknowledgment goes to my girlfriend Sarah Grebing for her support, patience and encouragement during my work on this thesis and for being a constant source of motivation.

I am also deeply thankful for the help and moral support of my friends over the last few years. Last but definitely not the least, I would like to thank my family, in particular my parents, for their support during my studies and throughout the course of this thesis and for always believing in me.

This work is partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008 and as part of the Software Campus program. I also would like to acknowledge funding by the Klaus Tschira Foundation.

Zusammenfassung

Deduktive Programmverifikation als Methode zum Nachweis von Korrektheitseigenschaften findet ihren Ursprung in den Arbeiten von Floyd und Hoare Ende der 1960er Jahre [Flo67; Hoa69]. Trotz der Weiterentwicklung der Spezifikations- und Verifikationskonzepte und dazu passender, praktisch anwendbarer Verifikationswerkzeuge ist die formale Programmverifikation bis heute kein gängiges Mittel der Qualitätssicherung in der industriellen Softwareentwicklung, sondern beschränkt sich oft auf kleine, sicherheitskritische Systemkomponenten. Ein Grund für den eingeschränkten Einsatz formaler Verifikation ist der hohe personelle Aufwand in der Spezifikation und Beweisführung.

Verbesserungen am Automatisierungsgrad von Theorembeweisern für Prädikatenlogik mit Theorien (wie Z3 [MB08]) in den letzten Jahren ermöglichten deren Einsatz in Programmverifikationssystemen und etablierten ein neues Interaktionsmuster, die sogenannte auto-aktive Verifikation [LM10]: Während üblicherweise nur die Anforderungsspezifikation in der Programmverifikation im Programmcode annotiert wird, erfolgt bei auto-aktiven Verifikationssystemen auch die sonst notwendige Benutzerinteraktion (etwa die Angabe von Lemmata) mit dem Beweiser mit Hilfe von Annotationen. Der Programmcode samt Annotationen wird nach erfolgter Annotierung in eine prädikatenlogische Formel übersetzt, deren Allgemeingültigkeit die Korrektheit des ursprünglichen Programms bezüglich seiner Spezifikation impliziert. Die Überprüfung der Allgemeingültigkeit dieser Formel erfolgt mit Hilfe eine SMT-Solvers wie etwa Z3.

Trotz eines erhöhten Automatisierungsgrades in der Beweissuche bleibt die Interaktion des Benutzers mit dem Verifikationssystem ein zentraler und arbeitsintensiver Bestandteil der Beweisführung. Während bisher der Aufwand in der interaktiven *Verifikation* begründet lag, ist durch die verbesserte Effizienz der Beweiser nun die Formulierung einer (auch für den Beweiser) geeigneten *Spezifikation* der Flaschenhals – sowohl für die Anforderungsspezifikation, als auch für die notwendigen Hilfsannotationen.

Die Basis der vorliegenden Arbeit bildet die Spezifikation und Verifikation eines komplexen, großen Softwaresystems – der industriell eingesetzte PikeOS Microkernel [SYS14; KW07] – im Rahmen des BMBF-geförderten Verbundprojekts Verisoft XT [Ver10]. Der erste Teil dieser Arbeit definiert dazu den Korrektheitsbegriff für den nebenläufigen Betriebssystemkern PikeOS als Simulationseigenschaft zwischen einer abstrakten Spezifikation und der Implementierung in C. Die Verifikation der so spezifizierten Korrektheitseigenschaft erfolgt mit Hilfe des unter anderem von Microsoft Research im Rahmen von Verisoft XT entwickelten Verifikationswerkzeugs VCC [Coh+09b]. Die auto-aktive Verifikationsmethodik des verwendeten Werkzeugs VCC erlaubt die für den Korrektheitsbeweis notwendige Spezifikation vollständig auf Quellcode-Ebene anzugeben. Exemplarisch wird an der Implementierung eines einfachen Systemaufrufs die verwendete Spezifikationsmethodik als Teil des Simulationsbeweises vorgestellt.

Ein Fazit der Arbeiten ist die prinzipielle Machbarkeit der Verifikation realistischer Software, die nicht explizit zum Zweck der Verifikation entsprechend entwickelt wurde – dies wird zum einen durch eine Spezifikationsmethodik ermöglicht, die flexibel genug ist, die gewünschten Eigenschaften hardwarenaher Programme auszudrücken, zum anderen durch eine Beweiskomponente, die mächtig genug ist, um Eigenschaften mittels auto-aktiver Verifikation nachweisen zu können. Gleichzeitig zeigt Verisoft XT auch die Grenzen der Anwendbarkeit deduktiver Verifikation in der Softwareentwicklung.

Die Fallstudie der Verifikation des Microkernels PikeOS ermöglicht dabei einen Einblick in die Gründe für den hohen Arbeitsaufwand und die Schwierigkeiten bei der Verifikation komplexer, nebenläufiger Systeme. Im zweiten Teil der Arbeit greifen wir einige der beobachteten Probleme auf und geben Vorschläge an, die den Spezifikations- und Verifikations*prozess* verbessern sollen:

Vollständigkeit des Verifikationssystems Die für einen Beweis mittels des auto-aktiven Verifikationswerkzeugs VCC benötigten Hilfsannotationen lassen sich bei genauer Betrachtung in verschiedene Klassen anhand ihrer Bedeutung für den Beweisprozess einteilen. Während ein Typ von Annotationen lediglich die Beweissuche verkürzt bzw. erst eine erfolgreiche Suche ermöglicht (etwa durch Lemmata), sind andere Annotationen unabdingbar für die *Existenz* eines Beweises (etwa Schleifeninvarianten). Diese Einteilung ist orthogonal zu den verschiedenen Annotationstypen (wie Zusicherungen und Methodenkontrakte) – so sind z. B. nicht alle Schleifeninvarianten essentiell für die Existenz eines Beweises. Weiterhin gibt es in der VCC-Spezifikationssprache keine syntaktische Unterscheidung zwischen Annotationen verschiedener Klassen.

Ein genaues Verständnis der verschiedenen Annotationstypen ist Voraussetzung für den erfolgreichen Einsatz auto-aktiver Werkzeuge. Ohne diese Kenntnis führen Änderungen an der Spezifikation nicht zu einem Beweis, solange auch nur eine essentielle Annotation fehlt. Wir schlagen daher die Einführung einer syntaktischen Unterscheidung vor, die es erlaubt, die Bedeutung von Annotationen kenntlich zu machen. Mit dieser zusätzlichen Information geht auch eine bessere Wartbarkeit der Annotationen einher.

Die Tatsache, dass es Annotationen gibt, die notwendig für die Existenz eines Beweises sind, führt weiterhin zu dem neuen Begriff der Annotations-Vollständigkeit, die an die Stelle der üblicherweise geforderten relativen Vollständigkeit eines Verifikationssystems tritt: wir nennen intuitiv ein System annotations-vollständig, wenn es für jedes Programm und jede dazu passende Spezifikation eine Menge von Hilfsannotationen gibt, sodass das System die Korrektheit des Programms bezüglich dieser Spezifikation ableiten kann (sofern das Programm die gegebene Spezifikation erfüllt).

Testen von Axiomatisierungen Die Korrektheit eines Programmverifikationssystems stützt sich zum einen auf die korrekte Implementierung der Schlussregeln des verwendeten Kalküls, bzw. auf die richtige Übersetzung des Programms samt Spezifikation in eine prädikatenlogische Formel im Fall von auto-aktiven Systemen wie VCC. Zum anderen ist die Korrektheit der Schlussregeln bzw. die Axiomatisierung der verwendeten Prädikate und Funktionen Grundvoraussetzung für die Korrektheit des Gesamtsystems.

Die Reichhaltigkeit der Progammier- und Spezifikationssprache hat eine entsprechend komplexe und umfangreiche Axiomatisierung zur Folge – im Fall von VCC umfasst die verwendete Axiomatisierung etwa 400 Axiome. Zu diesen vorgegebenen Axiomen erlaubt es die VCC-Spezifikationssprache zudem benutzerdefinierte Axiome anzugeben.

In einigen Fällen liegt der verwendeten Programmiersprache keine formale Semantik zugrunde, die einen formalen Korrektheitsbeweis der verwendeten Schlussregeln bzw. Axiome ermöglichen würde. Als Möglichkeiten der Qualitätssicherung bleiben damit etwa die manuelle Inspektion der Axiomatisierung, Vergleichsprüfung mit der Axiomatisierung anderer Werkzeuge oder manuell erstellte Testfälle. Während die ersten beiden Methoden mit großem Aufwand verbunden sind und sich daher schlecht zur Überprüfung eines in der Entwicklung befindlichen Systems eignen, können Testsuites automatisiert Fehler aufdecken und liefern somit einen ersten Hinweis auf die Qualität der Axiomatisierung.

Um die Aussagekraft der Ergebnisse einer Testsuite beurteilen zu können, bedarf es einer Qualitätsmetrik für Testfälle, die auch miteinbezieht wie gut ein Testfall die Axiomatisierung testet. Analog zur Definition der Testabdeckung in der Softwaretechnik definieren wir hierzu, basierend auf Vorarbeiten, den Begriff der Axiomatisierungs-Abdeckung als Testmetrik, um die Güte eines Testfalls bezüglich der Axiomenmenge erfassen zu können. Diese Metrik erlaubt es, bestehende Testsuites zu analysieren und systematisch zu verbessern.

Um die Zweckmäßigkeit dieses neuen Coverage-Begriffs zu untersuchen, bestimmen wir für zwei Verifikationssysteme die Abdeckung der mitgelieferten Testsuite. In einem weiteren Experiment werden zu einer dieser Testsuites weitere Testfälle hinzugefügt, die bisher nicht abgedeckte Axiome verwenden; die dabei aufgedeckten Fehler demonstrieren den Nutzen der Testmetrik.

Eine weitere Anwendung der Coverage-Berechnungen ist die Erstellung neuer Testfälle aus bereits existierenden – die Menge der durch einen Testfall abgedeckten Axiome ergibt, zusammen mit dem Testfall selbst, einen (präziseren) Regressionstest.

Durch die verbesserte Testmethodik ist eine schnellere Stabilisierung des Entwicklungsprozesses hin zu einer weitgehend fehlerfreien Axiomatisierung (die danach mit anderen Methoden überprüft werden kann) zu erwarten. Dies reduziert die Zahl der erforderlichen Nachbesserungen am Verifikationssystem und zugleich die Überprüfungen der mit einer inkorrekten Axiomatisierung erfolgten Korrektheitsbeweise.

Verbesserte Spezifikationssprache durch abstrakte Datentypen Neben der Mächtigkeit der zur Verifikation eingesetzten Beweiser beeinflusst die Wahl der unterstützten Spezifikationskonzepte und -sprache maßgeblich die Effizienz des Verifikationssystems. Bei auto-aktiven Verifikationswerkzeugen kommt der richtigen Wahl der Spezifikationssprache aufgrund der Tatsache, dass diese die einzige Interaktionsmöglichkeit mit dem Verifikationswerkzeug ist, eine besondere Bedeutung zu.

Wichtig ist dabei die Spezifikation auf einer geeigneten Abstraktionsebene angeben zu können – die passende Abstraktion ist dabei jeweils abhängig von der Anwendung. Im Rahmen des Verisoft XT-Projekts flossen die Erfahrungen in der Betriebssystemverifikation in die Weiterentwicklung der Spezifikationssprache ein, z. B. unterlag das zugrunde liegende C-Speichermodell mehreren Änderungen durch die VCC-Entwickler, um etwa die Spezifikation von Aliasing-Bedingungen effizient handhabbar zu machen [Alk+10c].

So erfordert die Spezifikation von gängigen Datenstrukturen (z.B. doppelt verkettete Listen) mitunter bereits erheblichen Annotationsaufwand. Dies legt den Schluss nahe, dass für diese Fälle die optimalen Spezifikationsmuster und -primitive noch nicht in der VCC-Spezifikationsmethodik vorhanden sind.

Um eine konzise und gleichzeitig verständliche Spezifikation für Operationen auf Datenstrukturen zu ermöglichen, schlagen wir vor, der VCC-Spezifikationssprache Primitive zur Definition von abstrakten Datentypen (ADTs) hinzuzufügen. Dabei kann auf eine Fülle von Vorarbeiten in anderen Gebieten, etwa algebraische Spezifikationssprachen wie CASL [Ast+02], zurückgegriffen werden.

Ausgehend von einer VCC-Spezifikation für eine einfache Datenstruktur zeigen wir exemplarisch die Spezifikation mittels der neu eingeführten Spezifikationsprimitive für abstrakte Datentypen.

Frühzeitige Rückmeldung fehlgeschlagener Verifikationsversuche Bevor der Benutzer eines auto-aktiven Verifikationssystems eine sinnvolle Rückmeldung des Systems zur Korrektheit einer Implementierung bezüglich seiner Spezifikation erwarten kann, sind eine Reihe von Hilfsannotationen notwendig – dies sind gerade die essentiellen Annotationen, deren Existenz zur Annotations-Vollständigkeit eines Systems gefordert ist. Für den Fall, dass das Programm seine Spezifikation nicht erfüllt, sind Teile der erstellten Hilfsannotation nach Korrektur der Fehler oft nicht mehr gültig. Bei einer negativen Rückmeldung des Verifikationssystems muss außerdem zwischen einer zu schwachen Hilfsannotation und einem tatsächlichen Fehler im Programm oder in der Anforderungsspezifikation unterschieden werden.

Weiterhin führen wechselseitige Abhängigkeiten zwischen den einzelnen Annotationen (etwa Kontrakte einer aufrufenden und der aufgerufenen Methode) dazu, dass bei der Spezifikation kein reiner top-down oder bottom-up-Ansatz möglich ist. Für die effiziente Verifikation eines komplexen Softwaresystems muss der Benutzer vielmehr in der Lage sein, den Ausgangspunkt der Verifikationsarbeiten frei wählen zu können – dabei müssen notwendigerweise Annahmen über die Eigenschaften der Programmteile getroffen werden, die mit der gerade zu verifizierenden Komponente in Beziehung stehen. Diese Annahmen werden in einem späteren Schritt gegen die konkrete Implementierung geprüft. Sind die Annahmen unzutreffend, müssen nicht nur die Annahmen selbst, sondern alle davon abhängigen Spezifikationen korrigiert werden.

In beiden genannten Fällen ist daher eine frühzeitige und präzise Fehlermeldung des Verifikationssystems von Vorteil. Wir demonstrieren dazu eine Methode, die Software Bounded Model Checking [CKL04] mit dem Prozess der annotationsbasierten deduktiven Verifikation verbindet und in der Lage ist, Fehler frühzeitig zu erkennen. Der dazu implementierte Prototyp liefert dem Benutzer konkrete Programmläufe zur Analyse.

1. Introduction

Correct operation of computer systems is becoming increasingly important in today's society – first and foremost affecting safety-critical and security-critical systems like control software for nuclear power plants, or systems storing and processing sensitive data like personal health records. At the same time, faults in these systems become more probable as the complexity of hardware and software components increases. In addition, with the ubiquity of (embedded) computer systems, most of them connected to the Internet, (automatically) exploiting many systems is a viable and apparently rewarding type of attack. As a result, correct functioning of computer systems also becomes a concern for regular users of personal computers, e.g., when entrusting bank account details to their favorite web browser.

Classic examples for disastrous outcomes of safety failures found in the literature include flight 501 of the Ariane 5 launcher [Lio+96], resulting in the self-destruction of the rocket due to an error in the control software, or the accidents involving the Therac-25 radiation therapy machine, administering overdoses of radiation to patients [LT93]. The state of computer security is no less serious: by exploiting vulnerabilities of computer systems, attackers regularly compromise security of systems and acquire a huge number of data records, ranging from email addresses to account data including passwords up to credit card details and detailed personal information (for two recent incidents of data security breaches of a large scale, see [She11; Fin13]). A prominent example for the severity of security issues in software systems is a vulnerability in the widely deployed cryptographic library OpenSSL [OSSL14], which remained undetected for two years [Cod14] before it was discovered in 2014 [MITR14]. This vulnerability, which has become known as the Heartbleed bug, can be exploited to expose sensitive data on a computer running OpenSSL.

To address these issues much effort is put into techniques to improve the quality of software and hardware, starting with precisely recording requirements and employing best practices of software engineering in the design and implementation phase. Many methods have been established that can provide certain guarantees for the resulting documents or implementations – manual audits or software testing are just two examples. In addition, various formal methods have found their way into the software quality assurance process – for instance model checkers to detect bugs in processors, static program analysis tools detecting common programming errors (like null pointer dereferencing in C), as well as techniques to check programs against user-provided explicit specifications (e.g., Microsoft's Code Contracts [FBL10; Log13], together with static analysis tools or the testing tool Pex [TH08]). Following the idea of software construction that is correct by design, frameworks such as the B-method [Abr96] provide techniques to refine an abstract specification of a program down to the concrete implementation.

One possibility to obtain a definite statement about the correctness of a program is to provide a rigorous logical proof that the program adheres to its given requirement specification. Deductive verification tools support conducting these proofs by mechanizing and automating (part of) the reasoning task. Efficiency and effectiveness of deductive program verification tools and methodologies has improved in recent years to the point where large scale verification projects are made possible by state-of-the-art verification systems – one example is the advent of an interaction pattern between the user and verification system called *auto-active* verification [LM10], implemented in tools like VCC [Coh+09b]. This type of user interaction follows the idea of a verifying compiler, one of the grand challenges in computer science as proposed by Hoare [Hoa03].

Providing the right user input is still a crucial and non-trivial issue for current verification tools in order for a program correctness proof to succeed. A recurrent topic of this thesis is the practicability of deductive verification techniques – in particular when applied to realistic computer systems, where many deficiencies of the tools and methodologies used become especially noticeable. In this thesis the verification of the PikeOS[1] microkernel [SYS14; KW07] carried out within the Verisoft XT project [Ver10] will serve as a case study for the specification and verification of a realistic software system.

1.1. Formal Methods in Practice: the Verisoft Projects

Designed as long-term research projects, Verisoft [IP08] and Verisoft XT [Ver10] started in 2003 with the goal of "pervasive formal verification of computer systems" [Ver10]. Funded by the German Federal Ministry of Education and Research (BMBF) with a project duration of four years for Verisoft and three years for Verisoft XT, partners from industry and academia developed formal verification methods that were then applied to (both industrial and academic) case studies, e.g., from the automotive or avionics domain. In Verisoft XT, a consortium of twelve project partners from industry worked together with six universities and employed formal methods to ensure correctness properties of several large case studies – demonstrating feasibility of formal verification as part of the software engineering process, also reflected in the project's motto "verification as an engineering science".

The correctness of a program rests on the correctness of all underlying layers of the computer system used – among others, from design of the processor at gate level, the correctness of the operating system up to the right assumptions about the behavior of the system's environment. To provide a *pervasive* formal correctness argument for an entire computer system was one of the goals in the first Verisoft project. In one of the sub-projects within Verisoft, a complete computer system (called "academic system") was implemented featuring an email client able to transmit (signed) mails as user-level demonstrator program [BHW06]. Below this top-level user space program, all necessary components of the academic system were developed during the project duration – i.e., among others, starting from a reduced instruction set processor (RISC), called VAMP [Bey+06], to a microkernel [Dör10] and operating system [Bog08], as well

[1]See `http://www.pikeos.com`

as a compiler from the high-level language C0 used on the upper implementation layers to the RISC instructions executed on the hardware [Lei08]. Correctness properties of the different implementations were verified with the help of interactive theorem provers, importing the properties established on the lower levels (like the operating system or compiler correctness). Verification frameworks, together with theories linking the different layers were also developed within Verisoft, the latter giving a solid foundation for the overall correctness argument of the whole academic system.

The pervasive correctness proof as one result of the first Verisoft project [Alk+10d] demonstrated that formal verification of a large and complex computer system is feasible – however, it also became apparent that the cost involved through interactive verification alone (even when reusing the acquired tools) would impede application of software verification in a similar setting in industry. One of the reasons for the huge verification effort was the degree of automation of the proof calculi – even optimized proof search strategies require a lot of user-provided supervision and hints, simply because of the size of the problems involved. Exemplary for the overhead incurred by user interaction is the verification of the seL4 microkernel within the L4.verified project [Kle+10], using the Isabelle framework, 200 000 lines of proof script were needed to prove 7500 lines of implementation correct [NIC11a].

To simplify verification of a software system of this size and complexity within Verisoft, the implementation of each component of the academic system was not tuned to good performance but rather adapted in some cases to simplify the proof task. As a consequence, when the components of the academic system were first integrated, update times of the user interface of the academic email client running on the full system stack were too slow to be usable in practice.

As a consequence, for the successor project Verisoft XT, different verification targets where chosen to advance application of formal verification techniques to real-world scenarios. Pervasive verification of the whole system stack was abandoned in favor of being able to handle a single, but larger and more complex component, i.e., system software in the form of virtualization platforms. Still, even by concentrating on one component in the system stack, specification and verification methodologies had to be scaled up to successfully complete the correctness proof – accordingly, one goal for Verisoft XT was to develop a program verification tool with a high degree of automation [Ver07]. A rationale for choosing system software as the verification target in Verisoft XT, Cohen et al. [CPS13] note that "[...] it represents a relatively small code base that is widely used, of critical importance, and hard to get right". In two sub-projects of Verisoft XT, existing implementations of system software used in industry (i.e., the Microsoft Hyper-V hypervisor, as well as the PikeOS microkernel by SYSGO) were to be specified and verified both using the VCC verification toolchain developed by Microsoft Research.

This thesis is based on the specification and verification of the PikeOS microkernel within the avionics subproject of Verisoft XT. In the context of this subproject, core parts of the embedded hypervisor PikeOS have been verified. The PikeOS kernel is part of a virtualization concept targeted at embedded real-time systems [SYSa] to support safety-critical and security-critical applications and is deployed in industry in areas such as automotive or avionics. The size of this microkernel is several orders of magnitude smaller compared to, e.g., the Linux kernel, with the amount of source code described by SYSGO as "less than 10,000 lines of code" [SYSc]. The exact line count depends on the architecture of the host system where PikeOS is deployed, as well as the counting metrics, and is not generally disclosed.

1.2. Current State of Deductive Verification

Deductive program verification has a long tradition, beginning with Floyd's work on "Assigning Meanings to Programs" in 1967 [Flo67] and Hoare's work introducing the eponymous formal system to reason about programs two years later [Hoa69]. A huge range of deductive verification methodologies and tools have emerged since – addressing challenges such as specification and verification techniques for several programming paradigms like object-oriented programming, concurrency, as well as properties beyond functional specification, e.g., in the form of information-flow properties.

Common to all these tools is the need for user interaction, not least due to the given theoretical limit of undecidability of the underlying problem, but also because of the limited efficiency of the automatic proof search of the systems. Towards a tool to prove the correctness of programs, Hoare put up the grand challenge of a *verifying compiler*, a tool that given specifications "[...] by types, assertions, and other redundant annotations associated with the code of the program" [Hoa03], provides a statement about whether the program meets its specification or not.

In recent years, deductive program verification tools have made significant progress, contributing to a realization of the vision of a verifying compiler. Full functional verification of individual functions written in real-world programming languages is practicable with reasonable effort. Verifying large and complex software systems is also feasible, as shown in the L4.verified and Verisoft projects using system software as the verification target [Kle+10; Bau+09b]. Naturally, verifying non-trivial software systems requires substantial effort due to the size of the system alone. In addition, however, even with modern specification methodologies and verification tools, verification of large software systems suffers from scalability issues. Like in the software development process – where the effort of implementing a complete system is more than the sum of implementing its isolated components – the effort of specifying and verifying a real-world system is more than the sum of verifying its components.

1.3. Contents and Structure of the Thesis

In this thesis we report on lessons learned from the Verisoft XT project, which we divide into two parts: the first is concerned with the specification and verification of a specific software system, in our case the PikeOS microkernel within the Verisoft XT project, as an example for the specification and verification of existing, real-world software (in contrast to verification targets adapted to simplify the proof effort). We will present the specification and verification methodology used to verify the correct operation of the PikeOS kernel and identify issues within the verification process using current, state-of-the-art verification tools and methodologies.

Based on our experience gained from conducting functional correctness proofs for parts of PikeOS within Verisoft XT, we discuss the issues encountered relevant for large-scale verification of real-world systems – together, these issues concern the whole software verification process: starting with the task of devising adequate specifications and turning them into appropriate annotations, to interpreting and incorporating feedback of the verification attempt, as well as organizing change management (both for the verification target and the verification tool). We then address three aspects of these problems in detail in the second part of this thesis to improve the verification process and to provide a better tool usability.

Some of these issues surprised us (e.g., the occasionally fragile interdependencies of annotations *within* a single module) while others were expected to occur from the beginning of the project on (e.g., complex interaction of the different components of a preemptible operating system). But even for the expected problems, their impact on the verification effort often deviated from our anticipations. For example, the difficulty to verify complex algorithms and data structures turned out to be of little concern in the project (as the implementation of PikeOS avoided this complexity for good reason).

The two main conclusions from our work in the Verisoft XT projects are: first, verification of complex concurrent software systems can be successfully done, however, even with state-of-the-art verification systems this incurs a substantial user effort. Secondly, given the power of modern verification tools, not verification but *specification* is the real bottleneck for large software systems – besides formalizing the requirement specification of a system, finding appropriate *auxiliary* specifications necessary to conduct a proof is a non-trivial and time consuming process (cf. Chapter 5 and Chapter 7). We argue that the latter insight applies not only to microkernels but to most non-trivial software system.

1.3.1. Part I – Verifying a Paravirtualizing Hypervisor

We start with a general introduction to virtualization (Sec. 2.1), continuing with a description of the PikeOS microkernel in Section 2.3, including a brief explanation of how PikeOS provides paravirtualization with the help of the underlying PowerPC hardware features (Sec. 2.2). We introduce several deductive verification tools in Section 2.4, including the VCC tool used to conduct the PikeOS case study and compare different verification approaches available.

For the verification of a (paravirtualizing) hypervisor, functional correctness is stated in terms of a simulation relation between the concrete implementation and the abstract specification of the system in terms of a transition system as given in Section 3.1, together with the general notion of different kinds of simulation relations.

Verification of PikeOS is then shown at a simple system call as one example for the concrete specification techniques used – both in a sequential setting and for the concurrent execution (Sec. 3.2). The latter version also includes how to handle preemption in the kernel using VCC mechanisms. The correctness of our verification approach depends on a set of assumptions, as well as further correctness arguments that need to be worked out and which are out of the scope of this thesis – in Section 3.3 we give a brief overview of these remaining building blocks in the theory justifying our verification technique.

We conclude this chapter with a comparison of our sub-project in Verisoft XT to other relevant system software verification projects of the past few years in Section 3.4, like the predecessor Verisoft, the Hyper-V sub-project of Verisoft XT or L4.verified.

Based on the experiences gained in the Verisoft XT project, we illustrate several challenges when specifying and verifying realistic, large-scale software (while the implementation of PikeOS is small compared to desktop operating systems like Linux, for specification and verification purposes the size of such a system is significant). With the help of examples, we demonstrate the issues occurring in large-scale verification and identify bottlenecks in the usability of the auto-active specification and verification framework in Chapter 4 – this is the basis of the second part, where we will then provide measures to improve on chosen issues from this enumeration to provide the user with better support in the overall specification and verification process.

We argue that some issues are inherent to the verification of imperative programming languages with heap state (e.g., framing). However, we will call attention to specification scenarios that are overly complex and can be improved upon that are orthogonal to these inherent problems – one example are simple heap data structures involving a considerable amount of specification overhead (cf. Chapter 8).

To find adequate (requirement) specifications for a system, it is indispensable to make use of existing documentation. One of the issues we identified is the mismatch between the level of abstraction used in the documentation and the abstraction level that is adequate in writing specifications – we will show at an example that the amount of information that is (necessarily) kept implicit in the documentation complicates simple re-use of the natural language specification on the formal annotation level.

We also found that change management in our context poses an additional challenge, concerning mainly changes in the verification methodology during the project duration. The amount of changes in the verification target we were able to permit in order to be able to handle the verification task had to be chosen carefully, though.

Besides these more general issues in verification of complex software, we briefly discuss consequences of the properties of our verification target. One example is the impact on the verification process caused by a given implementation that neither has been implemented with formal verification as first priority, nor (constrained by, e.g., certification reasons) was allowed to be changed considerably in this regard, as might have been useful.

As a result of the Verisoft XT subproject avionics, due to limited resources, we were not able to conduct a functional correctness proof for the whole PikeOS kernel. Instead, we restricted our attention to smaller parts of the kernel that implement core mechanisms relevant to establish safe and secure virtualization. Our contribution concerning the PikeOS verification task in Verisoft XT is twofold: firstly, we have adapted the specification techniques used in the Hyper-V project devised for the VCC tool for our operating system configuration with a preemptible kernel. Although the verification was completed only for a part of PikeOS, we claim that the specification methodology presented in this chapter should be possible to be extended to the remaining parts of the PikeOS implementation and, more generally, to other preemptible microkernel variants with a similar setup (e.g., systems with single processor). Secondly, with a more general impact of the findings on other verification projects, we have identified challenges in large-scale specification and verification as part of our work within Verisoft XT – we claim that the remaining bottleneck in modern auto-active verification systems is the *specification* rather than the verification task.

1.3.2. Part II – Improving the Verification Process

Based on the observations in the first part, we will first address the issue of finding appropriate (auxiliary) annotations, both by proposing new specification language features and by improving feedback of the verification system for failed verification attempts. Additionally, we investigate how to improve the quality of the implementation of a verification system by testing correctness of the axiomatization part. The common goal of the proposed techniques is to improve on the established processes involved in verification, i.e., most notably (a) to reduce the overall user effort in finding the right set of annotations for a given program and its requirement specification, but also (b) to support the verification tool developer by exposing errors in the system, thus reducing the amount of iterations needed to obtain a stable tool implementation – as a consequence, fewer changes in the verification tool may result in fewer rectifications in existing specifications to re-establish failed proofs. In the following, we briefly list our contributions towards these goals.

Identifying the Different Types of Annotations One prerequisite for being able to improve the specification language and also the verification process is to be aware of the different purposes of annotations for verification tools. Usually, annotations are differentiated between requirement specifications (a contract for the externally visible behavior of the specified component) and auxiliary annotations (which are simply needed to complete the correctness proof). We discuss in Chapter 5 that in typical annotation-based verification systems with a toolchain architecture (like VCC), we can moreover distinguish between *two* kinds of auxiliary annotations, warranting introduction of the notion of *essential* annotations.

Improving Trust in Verification Results Correctness of VCC, as all other deductive verification tools, both relies on correct implementation of the tool and on a sound and adequate axiomatization. While there are rigorous methods to ensure the correctness of an axiomatization, most are time consuming and some methods are also not applicable to all axioms in the axiomatization. In addition, the effort involved in proving axioms to be correct w.r.t. their intended semantics might cause that only a set of "core" axioms of the calculus are treated in this manner – user-defined sets of axioms for a specific problem domain given for one particular proof obligation are unlikely to receive the same attention.

To improve the trust in the axiomatization part of verification tools, we propose testing the axiomatization in Chapter 6, offering an alternative to improve the axiom set which also can be integrated in regression tests. A coverage measure is defined that allows estimating to which degree a set of test cases uses all given axioms. Axiomatization coverage, together with a second metric (the *selectivity* of a test case) can then be used to assess and improve the quality of existing test suites. We examine evolution of VCC's test cases over time w.r.t. axiomatization coverage, compare results with those of other verification tools like KeY and demonstrate at a small case study that increasing coverage indeed helps to reveal errors in the axiomatization.

Improving the Verification Process Coming up with a system specification is a non-trivial and iterative process. Usually, a lot of auxiliary annotations have to be provided by the user before a meaningful verification attempt of a functional property is possible. Using a combination of software bounded model checking, together with deductive verification allows for an early and precise response for the verification engineer (Chapter 7).

Improving the Specification Language Conducting a large verification case study allows identifying recurring specification patterns for closely related problems and to assess utility of introducing special support in the specification language for such patterns. The benefit of well-chosen new specification keywords or mechanisms is to obtain a more concise and elegant way to handle common specification cases: one prominent example are specifications based on abstract data types. We suggest in Chapter 8 to use an existing formalism, the Common Algebraic Specification Language (CASL), as basis to simplify annotations for typical data structures and illustrate at an example how the VCC specification relates to its ADT counterpart.

1.4. Contributions

The contributions of this thesis are, in the order they appear in this work:

Part I – Deductive Verification of an Industrial Microkernel

Chapter 3: A verification methodology for the preemptible paravirtualizing microkernel PikeOS, based on the specification and verification techniques developed as part of the Verisoft academic system and the Verisoft XT Hyper-V project.

Chapter 4: A detailed description of issues of annotation-based specification impeding application of deductive verification as part of the software engineering process of real-world applications, based on insights gathered during the PikeOS case study.

Part II – Improving Deductive Verification for Real-World Application

Chapter 5: A refined terminology to distinguish the different types of auxiliary annotations accompanying tool-chain-based verification tools,

Chapter 6: a method to increase trust in verification tools by testing the axiomatization, based on previous work by Wagner [BW10],

Chapter 7: a technique to gain earlier feedback in the specification process, identifying errors in specification or implementation using bounded model checking and

Chapter 8: a proposal for more concise and elegant specifications with the help of established formalisms for abstract data types.

1.5. Previously Published Material

Most of the material presented in this thesis has been published before. Here, I would like to give an account of which parts of this thesis are either (with some smaller modifications) identical to previous publications or have been extended to address recent experiments or research results.

Section 7.1.6 ("Bounded Software Verification") was written by the author of this thesis and is published in the work by Beckert et al. [Bec+14b]. For the verification of PikeOS with VCC in Section 3.2.2 ("Verifying PikeOS System Calls – Sequential Execution"), 3.2.3 ("Concurrency") and a smaller part of 3.3 ("Ingredients for Pervasive Correctness"), previous results and texts of the author have been published in work by Baumann et al. [Bau+09b; Bau+10]. Chapter 4 ("Lessons Learned from PikeOS Verification") has been largely written by the author and is, in parts, contained in the publication by Baumann et al. [Bau+12], with extended examples for the issues presented. Chapter 5 ("The Auto-Active Verification Paradigm") is the result of collaboration with Prof. Dr. Bernhard Beckert and Dr. Vladimir Klebanov and is identical to parts of our prior publication [BBK12] – also Chapter 8 ("Specification Using Abstract Data Types") is an (extended) version of the same publication. Chapter 6 ("Improving Trust in Verification Systems") extends our prior publication by Beckert et al. [BBW13a; BBW13b] (which in turn is based on first results by Wagner [Wag09]) with a significantly enhanced description and analysis of the coverage experiments performed by the author. Finally, Chapter 7 ("Improving Feedback for Verification") is based on material (to a large extent) written by the author from the publication by Beckert et al. [Bec+12], further extended by additional experiments and analyses.

The following list of publications contains all those papers I co-authored that are relevant for and used within this thesis, grouped by topic and sorted by year of publication. Of these twelve research papers, eight are peer-reviewed publications to workshops and conferences – the others include publications that appeared in proceedings of industrial conferences, written in collaboration with our project partner SYSGO during the Verisoft XT project.

In addition to the papers listed below, another more recent research topic I pursue is the verification of properties of voting schemes [Bec+14a; Bec+14b] – although related to the subject-matter of this thesis, it is not directly relevant for the topics presented and material from these publications is thus not included here.

PikeOS Verification

2009 C. Baumann, B. Beckert, H. Blasum, and T. Bormer. “Better avionics software reliability by code verification”. In: *embedded world Conference*. (Nuremberg, Germany). Ed. by M. Sturm. WEKA Fachmedien GmbH, Mar. 2009. ISBN: 978-3-7723-3798-7. URL: http://formal.iti.kit.edu/~bormer/pub/embeddedWorld2009.pdf

C. Baumann and T. Bormer. “Verifying the PikeOS microkernel: First results in the Verisoft XT avionics project”. In: *Doctoral Symposium on Systems Software Verification (DS SSV 2009)*. (Aachen, Germany, June 22–24, 2009). Ed. by R. Huuck, G. Klein, and B. Schlich. Aachener Informatik-Berichte AIB-2009-14. RWTH Aachen University, June 2009. URL: http://aib.informatik.rwth-aachen.de/2009/2009-14.pdf

C. Baumann, B. Beckert, H. Blasum, and T. Bormer. “Formal verification of a microkernel used in dependable software systems”. In: *28th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2009)*. (Hamburg, Germany, Sept. 15–18, 2009). Ed. by B. Buth, G. Rabe, and T. Seyfarth. Vol. 5775. LNCS. Springer, Nov. 2009, pp. 187–200. DOI: 10.1007/978-3-642-04468-7_16

2010 C. Baumann, B. Beckert, H. Blasum, and T. Bormer. “Ingredients of operating system correctness”. In: *embedded world Conference*. (Nuremberg, Germany). Ed. by M. Sturm. WEKA Fachmedien GmbH, Mar. 2010. ISBN: 978-3-7723-1012-6. URL: http://formal.iti.kit.edu/~bormer/pub/embeddedWorld2010.pdf

2011 C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev. “Proving memory separation in a microkernel by code level verification”. In: *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW 2011)*. (Newport Beach, California, Mar. 28–31, 2011). Mar. 2011, pp. 25–32. DOI: 10.1109/ISORCW.2011.14

2012 B. Beckert and T. Bormer. “Lessons learned from microkernel verification”. In: *Pre-Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)*. (Bamberg, Germany). Ed. by G. Lüttgen and S. Merz. 2012

C. Baumann, B. Beckert, H. Blasum, and T. Bormer. “Lessons learned from microkernel verification: Specification is the new bottleneck”. In: *Proceedings of the Seventh Conference on Systems Software Verification (SSV 2012)*. (Sydney, Australia, Nov. 28–30, 2012). Ed. by F. Cassez, R. Huuck, G. Klein, and B. Schlich. Electronic Proceedings in Theoretical Computer Science 102. 2012. DOI: 10.4204/EPTCS.102.4

Annotation-based Specifications

2011 B. Beckert, T. Bormer, and V. Klebanov. "Improving the usability of specification languages and methods for annotation-based verification". In: *9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*. (Graz, Austria, Nov. 29–Dec. 1, 2010). Ed. by B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue. Vol. 6957. LNCS. Springer, 2012, pp. 61–79. ISBN: 978-3-642-25270-9. DOI: 10.1007/978-3-642-25271-6_4

Improving the Verification Process

2012 B. Beckert, T. Bormer, F. Merz, and C. Sinz. "Integration of bounded model checking and deductive verification". In: *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2011), Revised Selected Papers*. (Turin, Italy, Oct. 5–7, 2011). Ed. by B. Beckert, F. Damiani, and D. Gurov. Vol. 7421. LNCS. Springer, 2012, pp. 86–104. DOI: 10.1007/978-3-642-31762-0_7

Testing the Axiomatization of Verification Tools

2010 T. Bormer and M. Wagner. "Towards testing a verifying compiler". In: *Formal Verification of Object-Oriented Software, Papers presented at the International Conference*. (Paris, France, June 28–30, 2010). Ed. by B. Beckert and C. Marché. Vol. 2010-13. Karlsruhe Reports in Informatics. Karlsruhe Institute of Technology, Technical Report, 2010

2013 B. Beckert, T. Bormer, and M. Wagner. "A metric for testing program verification systems". In: *Seventh International Conference on Tests and Proofs (TAP 2013)*. (Budapest, Hungary, June 16–20, 2013). Ed. by M. Veanes and L. Viganò. Vol. 7942. LNCS. Springer, 2013, pp. 56–75. DOI: 10.1007/978-3-642-38916-0_4

B. Beckert, T. Bormer, and M. Wagner. "Heuristically creating test cases for program verification systems". In: *10th Metaheuristics International Conference (MIC 2013)*. (Singapore, Aug. 5–8, 2013). 2013

Part I.

Deductive Verification of an Industrial Microkernel

2. Preliminaries

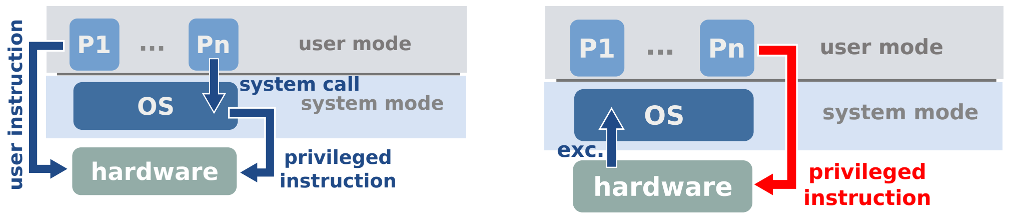
Contents

2.1. System Virtualization

2.1.1. Operating Systems

Apart from the earliest computer systems, sharing available computing resources provided by a single machine between multiple programs (possibly by different users) to use the hardware to full capacity has been a recurrent topic. This, and the ability to abstract from particularities of the underlying hardware gave rise to operating systems, starting with the first inception in the 1950s to a concept already similar to approaches known today in the following decade [Doe11], scheduling several processes to run on the processor and managing other resources like memory, such that each process is able to achieve its task without (unintended) interference by other programs on the same machine.

This separation of processes entails that there are operations that are not available to all programs, e.g., changing which parts of memory a program is allowed to access, but rather reserved to the operating system. Hardware supports this separation of privileges by providing different execution modes, usually at least a system mode with full access to the processor's instruction set and a (restricted) user mode. Execution of user mode instructions issued by user programs are run as-is on the hardware, system mode functionality is accessed through system calls of the operating system (see Fig. 2.1a). If a user program tries to execute instructions reserved to system mode, the hardware responds with an exception which can be handled by the operating system (Fig. 2.1b), which in turn may take appropriate actions (e.g., terminate the offending process).



(a) Execution of user instructions and access to system mode functions.

(b) Handling of illegal execution of system mode instructions by a user process.

Figure 2.1.: Operating system privilege separation.

2.1.2. Full System Virtualization

For safety-critical and security-critical computer systems, a general concern is the size of the trusted computing base. Reducing the size and amount of components that need to be trusted to guarantee the intended properties of a system is one option to reduce the probability of errors and also helps in making (formal) analysis techniques feasible. While the size and complexity of the virtualization layer adds to the size of the guest system, virtualization allows removing untrusted software from the trusted computing domain by separating subsystems in self-contained virtual machines.

In contrast to "regular" operating systems, full system virtualization provides the illusion of several instances of the same hardware to be available for running software (the *guest* systems) in isolation. According to Silberschatz et al. [SGG08], support for virtual machines goes back to 1967 to IBM's CP/67 operating system intended to run on the IBM 360/67 mainframe, providing programs the illusion of executing on isolated System/360 hardware. Virtualization allows consolidating multiple, isolated guest operating systems together with their applications on a single machine to achieve possibly higher capacity utilization of computing resources. In addition, it helps in testing and developing system level software – the user may switch between different systems and configurations without compromising the host system.

The hardware simulated in full system virtualization is (mostly) identical to the real machine (the *host* system) the virtualization layer and guest systems operate on. A hypervisor (also called virtual machine monitor) provides this functionality to the guest systems running on top – normal operating systems that run *unmodified* as guest systems provide the features described earlier. Figure 2.2a shows this full system virtualization setup.

There are different techniques how to achieve this – one technique available on every host machine is to emulate execution of each guest instruction by, e.g., interpreting the guest code. However, while this method provides the intended illusion for the guest system running on an unshared copy of the hardware, it does not live up to the expectations of system virtualization as described by, e.g., Popek and Goldberg [PG74], which require execution of programs on the virtualized system to be efficient, compared to the real execution speed of the host hardware (besides efficiency and equivalence of the simulated machine, the third condition proposed by Popek and Goldberg is that the "virtual machine monitor is in complete control of system resources"). As a consequence, for all but a few of the instructions issued by a guest, the virtualization component should not be needed to interfere as this would incur additional overhead, but rather most instructions have to run directly on the host system.

Some instructions of the host system are not allowed to be executed unmodified by a guest, as these instructions might interfere with execution of other guests or the virtualization layer itself. In these cases, the virtualization component has to provide mechanisms to simulate appropriate behavior of the virtualized system. One example are privileged instructions – as explained above, if executed in user mode this type of instructions transfers control to the operating system (running in system mode).

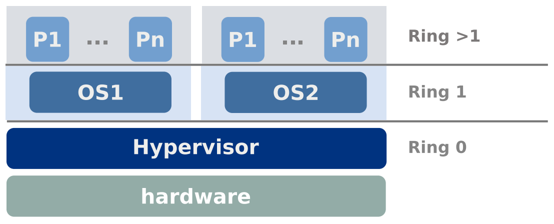
Popek and Goldberg call the instructions in question *sensitive* and give a sufficient condition on the host hardware to be able to efficiently virtualize a computer system: the set of sensitive instructions has to be a subset of the set of privileged instructions. This requirement enables to treat sensitive instructions by appropriate handlers in the operating system, as all sensitive instructions can be intercepted by the operating system.

A prominent example found in the literature of an architecture that does not meet this requirement is the x86 architecture, where some sensitive instructions executed in user mode have a different effect compared to system mode but do not cause a trap when executed in user mode [SN05]. Hence, the method of using traps alone is not sufficient to virtualize the x86 architecture – techniques such as ad hoc binary translation, introduced in a commercial product by VMWare, are used for problematic instructions by rewriting them to sequences of instruction of the host hardware [AA06]. In contrast to x86, the Power architecture (as used in our setup for the PikeOS microkernel) meets the virtualization criteria of Popek and Goldberg and can thus be virtualized using traps [Mit+13].

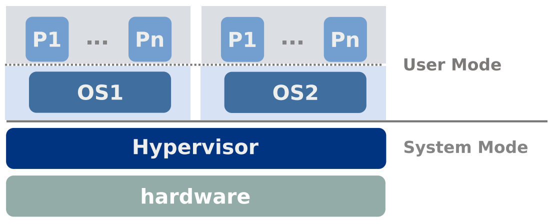
As per definition of full virtualization, the different instances of the hardware all offer the same separation of system and user mode which allows running multiple operating systems on the same physical hardware. Also, any available memory management unit (MMU) of the hardware has to be replicated – e.g, in the case of page table based memory management, the translation from guest virtual memory to physical memory takes another indirection and is supported by *shadow page tables* managed by the virtual machine monitor. In the case of full virtualization, physical hardware may facilitate this with additional privilege levels (e.g., as in the x86 architecture, called "rings" and shown in Figure 2.2a), together with MMUs providing this two-level translation as part of the hardware. With several privilege levels available, the virtualization platform can be executed at the highest privilege level, managing the different instances of virtualized hardware by providing an interface to the guest systems via *hypercalls* (e.g., providing functions to create new virtual machines).

There are two variants of full virtualization: Type-1 hypervisors run directly on the host hardware, while type-2 hypervisors make use of an operating system, providing abstractions of the underlying machine.

Today, many general purpose processors support virtualization in some form (e.g., as already mentioned, MMU virtualization support or several layers of increasing privilege). For desktop CPUs, Intel added the virtualization technology VT-x to some model ranges, whereas AMD's virtualization support is called AMD-V. The VT-x technology supports virtualization by adding a new, less privileged mode, orthogonal to the protection



(a) Full virtualization using multiple privilege levels.



(b) Paravirtualization setup using two privilege levels.

Figure 2.2.: Virtualization options.

rings provided by the x86 architecture – the guest system can then be executed in this mode, using all available protection rings, while the virtualizing operating system can supervise the guest [SN05]. A similar technique is employed in the AMD-V virtualization extension, though through a different interface incompatible with VT-x.

2.1.3. Paravirtualization

Recreating the effect of sensitive instructions for the virtualized guest systems can be costly when aiming for full system virtualization. Another virtualization option is to deviate from the one-to-one copy of the underlying host system and to provide a simpler interface to guest systems – this is the approach taken with *paravirtualization*. Figure 2.2b shows one particular paravirtualization setup with the guests, including the operating systems, running in user mode. Execution of privileged instructions both by user processes ($P_1, \ldots, P_n$ in Figure 2.2b) and by the operating systems ($OS_1, \ldots, OS_n$) now incur a trap. Thus, in contrast to full virtualization, where the guest systems do not have to be altered to be able to run on the virtual system, guests of a paravirtualized machine need to be modified to use the specific API of the underlying operating system. For some applications this is an obvious drawback, however, the opportunity to provide a new API allows abstracting from the hardware and to provide a simple interface to guest systems – porting guests to other hardware is then trivial, as long as the virtualization platform is available for the architecture. A good example for this kind of benefit, as given by Smith and Nair [SN05], is the Java virtual machine (although in the context of process-level virtualization rather than system virtualization).

Again, as in full virtualization, most instructions of a guest process are executed as-is on the host system, to make virtualization efficient, and only some functionality uses the API provided by the operating system.

The weaker simulation criterion of paravirtualization allows for a variety of possible implementations of the virtualization layer. The approaches to implement a paravirtualization hypervisor differ, among others, in the amount of operating system functionality provided by the hypervisor and in turn in the amount of changes necessary to be able to run existing operating systems as a guest. One sample operating system architecture suitable to achieve paravirtualization, together with a moderate size of code to be trusted are microkernels, as demonstrated by, e.g., L4.verified [Kle+10] and PikeOS [SYS14; KW07].

An example for a hypervisor providing full virtualization for the x86-64 architecture is Microsoft's Hyper-V, also a verification target in a sub-project of the Verisoft XT project. The microkernel PikeOS, with the strict separation of user processes in disjoint partitions can be ranked amongst the paravirtualizing hypervisors. Concerning type-2 hypervisors, one example is QEMU [Bel05], featuring the ability to emulate different architectures – dynamic recompilation is used to simulate the effect of instructions not available for the host system.

2.2. PowerPC

In this section, we give a brief overview of the features of the PowerPC hardware, focusing on properties of the PowerPC architecture relevant for the operation of a virtualization kernel like PikeOS. In particular, as our verification setup only considers one particular hardware configuration the kernel runs on, we instantiate abstract descriptions of the PowerPC architecture by concrete attributes of the particular platform – i.e., a 32-bit PowerPC architecture implemented on Freescale's MPC5200 platform featuring a G2_LE core.

The G2_LE core is a single core, superscalar reduced instruction set computing (RISC) processor for embedded devices, based on the PowerPC MPC603e design by Motorola/Freescale [Fre03]. System software operation is supported by two execution modes, user and system mode. Besides the 32 general-purpose registers (GPRs) available in both modes, system mode has access to special-purpose registers (SPRs), as well as additional instructions and resources like memory.

Whereas the specific PowerPC CPU used as hardware platform in Verisoft XT does not provide special features facilitating system virtualization, as mentioned in Section 2.1, the PowerPC architecture fulfills the Popek and Goldberg virtualization criterion.

The following description is a summary of the G2 PowerPC Core Reference Manual [Fre03] and the PowerPC Programming Environments Manual [Fre05].

2.2.1. Memory Management

Besides direct access to physical memory without address translation, the PowerPC hardware supports two address translation schemes – a page based memory mapping (with 4 KiB-sized pages), as well as mapping larger, continuous memory blocks (128 KiB to 256 MiB) through a mechanism called block address translation (BAT).

In each case, a 32-bit effective address (EA) generated by the CPU for, e.g., a data load or store operation is translated to a 32-bit physical address. Using the BAT mechanism, the translation is given by supervisor-level BAT register pairs storing the start and length of a memory region in the effective address space, together with the location of the corresponding (continuous) region in physical memory.

The page-based translation mechanism works by first translating the 32-bit EA to an 52-bit virtual address by combining the EA with the content of one of the 16 supervisor-level segment registers (selected using parts of the EA).

In a second step, the resulting 52-bit virtual address is converted to the 32-bit physical address used to access memory. First, the virtual address is looked up in a translation lookaside buffer (TLB), a cache for the mapping from virtual to physical addresses. If no appropriate entry is found in this cache, a trap to a handler in the operating system is taken to locate the corresponding entry in the page table in main memory which is placed in the TLB afterwards. Finding the right entry in the page table in memory has to be done in software, with some support by the memory management unit (MMU).

This translation scheme of using another indirection of memory segments requires only one page table per system instead of one page table for each process – process isolation in memory is achieved by choosing values of the segment registers such that different processes are mapped to different (disjoint) portions of virtual memory.

Besides address translation, the MMU also provides memory access protection with different granularities, restricting write access or denying execution of instructions for given memory locations. This mechanism may be used, e.g., by the operating system to disclose kernel-level data structures as a read-only memory page, while the kernel itself is allowed to access the same locations also for writing.

2.2.2. Interrupts

Another central part of the interplay between hardware and operating system are interrupts, allowing the system software to react upon events caused by user processes or the hardware. These events range from user processes invoking kernel functionality by issuing a system call instruction, to interrupts on certain memory accesses, enabling memory management and resource separation, as well as timer-based interrupts to be used for, e.g., process scheduling.

Handling an interrupt involves the following steps as defined by the PowerPC operating environment architecture (OEA): each interrupt is statically assigned a physical memory address at which control is transferred to by the hardware if the interrupt occurs (referred to as "taking" the interrupt) and where the system software has to place the interrupt handling routines. Starting with the execution of the first instruction of an interrupt handler, system mode is set and two special purpose registers are updated by the hardware providing further information about the machine context the interrupt occurred in.

The order in which several simultaneously occurring interrupts are taken depends on each interrupt's priority, as given in the OEA documentation. In addition, some asynchronous interrupts are maskable by setting a bit in a supervisor-level register. Masked exceptions recognized by the hardware are taken after the mask bit is cleared.

The OEA distinguishes between "instruction-caused" (i.e., synchronous) and "system-caused" (i.e., asynchronous) interrupts and further, for the synchronous interrupts, between precise and imprecise variants. For precise interrupts, when entering the handler, the exact address of the instruction relevant for the exception cause is passed to the handler – this guarantee is not made for imprecise interrupts. Examples for precise interrupts include the system call as well as the data storage interrupt; the only imprecise class of interrupts defined by the OEA concerns floating point operations. Interrupts that are system-caused are, e.g., system reset, external interrupts or the decrementer interrupt – the last two of these asynchronous interrupts are maskable.

In the system call handler other interrupts may occur – while proper implementation of the system software can ensure no further instruction-caused exception appears, asynchronous interrupts can by definition not be ruled out. To ensure that the interrupt handler can perform its task, the OEA states that all system-caused interrupts are masked when the interrupt is taken, except those that require unconditional and immediate action like system reset.

Interrupt handling is concluded with executing the return from interrupt (`rfi`) instruction, transferring control back to the location causing the interrupt.

2.3. PikeOS – An Industrial Microkernel for System Virtualization

The main object of consideration in the avionics sub-project of Verisoft XT, the PikeOS kernel[1] (see `http://www.sysgo.com/`), is a microkernel that is part of a virtualization concept targeted at embedded real-time systems [SYSa]. It acts as a paravirtualizing hypervisor to support safety-critical and security-critical applications and is deployed in industry in areas such as automotive applications or avionics, e.g., it is used in the Airbus A350 XWB [SYS08]. As such, it has been developed with certification in mind – e.g., the kernel allows certification according to the avionics safety-standard DO-178B [RE92; SYSd]. The features of PikeOS, being part of highly safety-critical applications and having a manageable code size (partly due to the microkernel architecture) made it a good choice for (the still challenging and costly) deductive verification, at the same time its deployment in existing industrial applications made for a highly relevant case study.

Below, we give a brief overview of features and implementation details of PikeOS. A more detailed description and an account of the development history of PikeOS, also compared to the L4 family of microkernels is given by Kaiser and Wagner [KW07].

Overview The implementation of PikeOS is available for multiple system architectures and platforms (PowerPC, ARM, x86, MIPS, SPARC V8/LEON, V850 and SH-4, according to [SYSb]). For verification purposes in the Verisoft XT project, the configuration has been fixed to one particular setup, a variant of the G2 PowerPC core running on the MPC5200 board (both by Freescale). In the beginning of the Verisoft XT project, only a single-core version of PikeOS was available – a multicore version of the code base was work in progress (which is now available). In the remainder of the thesis, the descriptions of PikeOS refer to the single-core version – also, we worked on a development snapshot of the PikeOS kernel as verification target within the Verisoft XT avionics sub-project, thus some details about PikeOS provided here may differ from the currently released product version.

In order to separate processes from each other, PikeOS provides partitioning measures, e.g., for memory resources and CPU time. The first, resource partitioning, guarantees that software applications operating on top of the hypervisor in one resource partition

[1]Further only called "PikeOS".

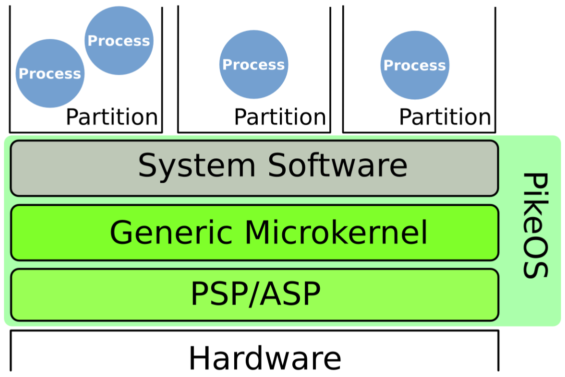


Figure 2.3.: A typical PikeOS virtualization setup featuring multiple isolated guest systems. (Modified version taken from [Bau+09b])

cannot read or write memory of other systems in different partitions (unless memory is explicitly shared across partitions). Time partitioning, on the other hand, allows for running systems depending on real-time functionality – with proper configuration, time-critical processes are guaranteed to be run at user-defined periodical intervals and get a sufficient share of the processor to run even despite possibly other ill-behaved processes consuming CPU time.

The Architecture of PikeOS The implementation of PikeOS is structured into several layers, amongst other reasons, to simplify porting the whole kernel to other architectures and boards. As shown in Figure 2.3, the lowest layer of PikeOS is the platform support package (PSP), which abstracts from details of the specific board used (in our case the MPC5200) and is also responsible for low level system initialization. One example of platform-specific functionality is the Interrupt Controller, which routes different interrupt sources to the CPU pins and is driven by the PSP. Besides the PSP, another lower-level part of PikeOS is the architecture support package (ASP), providing a processor abstraction including the processor context, exception handling and memory management. On top of these two packages another layer, more abstract in terms of hardware dependencies, completes the kernel implementation and establishes concepts such as tasks and threads. Necessarily, the hardware-related functionality, as well as performance-critical parts of the kernel are implemented in assembly language, while the majority of the kernel is written in C.

Having to treat only one specific hardware configuration, as mentioned above, together with the separation of parts of the hardware-related code in the PSP and ASP allows reducing the size of code base that is to be analyzed and verified within Verisoft XT. Compared to monolithic kernels like Linux, the implementation of a microkernel like PikeOS is several orders of magnitude smaller. In addition to simplify certification and verification tasks, this small size also helps in keeping the trusted computing base of the system to a minimum.

The PikeOS kernel implements a mechanism to manage threads and tasks and to control and restrict interaction of tasks using time and resource partitioning. Above the kernel layers, a special task running in user mode, the PikeOS System Software (PSSW) is used to manage setup of the guest operating systems to be virtualized. As mentioned in Section 2.1, guest systems running on top of a paravirtualization host have to be modified to make full use of the features of the hardware, as the guest systems are not able to use privileged mode to perform their work. Besides the PikeOS kernel functionalities, the PSSW establishes another API guest systems may use, e.g., providing a file system layer that manages requests to so-called file providers (which also are processes running in user space).

For system setup, the PSSW task needs to, e.g., setup memory mappings, create tasks or establish resource partitions – while most of the guest operating systems only have to be allowed to perform a subset of these actions. Therefore, a set of rights is associated with each task, e.g., restricting abilities to use PikeOS system calls or specifying which other task can be communicated with via inter-process communication (IPC). As the PSSW is granted the most extensive permissions of all user space programs to perform its work, it is also among the trusted components of the PikeOS virtualization platform. The PSSW, together with the kernel functionality, e.g., allow for ARINC 653 conformant partitioning. For our purposes of verifying a paravirtualizing hypervisor, however, the PSSW was not part of the object under consideration, as the kernel already provides the core mechanisms necessary to establish isolated guest systems.

The PSSW performs the setup of the guest operating systems according to user-provided, static configuration files – as a consequence, many parameters of the system setup are fixed throughout the system operation after the initialization phase.

PikeOS was originally designed following the concepts of the L4-family of microkernels in mind – however, since then, as development focused on real-time embedded systems, only parts of this ancestry remain [KW07]. The paradigm of the microkernel architecture to transfer functions normally residing in privileged mode in a monolithic kernel to userspace processes is of course also found in PikeOS. As in L4-based kernels, IPC plays an important role in PikeOS, not only for communication of user level processes among each other, but also, e.g., for handling exceptions in user space.

Resource Partitioning and Process Isolation One distinctive feature of the PikeOS kernel is resource partitioning, providing the foundation for security (by eliminating unintended interference between different domains), as well as contributing to consistent behavior of the user processes (as the system integrator is able to make provisions for enough resources on a partition basis). On an abstract level, with tasks being associated with an address space, resource partitions can be seen as collections of tasks and accordingly their address spaces [Bau+11].

As in every other operating system there are two kinds of memory allocations in PikeOS: (a) user memory used for the operation of guest processes and (b) kernel memory, used for, e.g., bookkeeping of information about threads, tasks or resource partitions relevant for correct kernel functionality.

In the following description of the PikeOS memory management and resource partitioning features, for simplicity we assume kernel initialization to be completed and omit the possibility of transferring memory mappings via IPC. The resulting state after initialization contains the PikeOS System Software as root task (in the first resource partition).

Physical memory for user space programs is allocated and mapped to virtual addresses by the PSSW, according to a static configuration file provided by the user. Separation of address spaces thus depends not only on correctness of the kernel or PSSW but also on correct configuration according to this file.

The system call functionality provided by the PikeOS kernel to allocate physical memory and create the initial memory mappings may only be used with the right abilities of the invoking thread. In our configuration, the only task with this ability is the PSSW, which, as a trusted component of the virtualization platform, does not grant this ability to user processes. As a result, further allocation of memory by user space programs is impossible.

Considering memory mappings, another mechanism prevents creation of arbitrary mappings which would cause unintended information flow: tasks may only change mappings of child tasks and only to their own valid address space. As the PSSW is the common ancestor of all user tasks, it is allowed to change mappings of any user task.

Correct operation of user space partitioning naturally relies on the cooperation of memory management functionality in PikeOS and the MMU of the PowerPC hardware. With proper configuration of the hardware, the MMU takes care of disjointness of the virtual address spaces of the different PikeOS tasks. As described in Section 2.2, the concrete CPU used in our verification target setup provides only little support for memory management, e.g., walking page tables to find a memory mapping has to be done in software. The functionality of PikeOS related to user-space memory management includes the system call interface to map and unmap memory pages, as well as switching address spaces when scheduling a thread not contained in the currently active task.

Setting the right address space when changing tasks requires to update the segment registers of the G2_LE core – as the content of the segment registers provide the upper 24 bit of the virtual address in the address calculation, having an injective function from tasks to segment register values ensures disjointness of virtual address ranges.

When another layer that stores mapping information in the form of a page directory is used, as described by Mackerras for the PowerPC/Linux kernel [Mac03], changing the memory mappings of a task is more involved: In addition to invalidating TLB entries for the changed mappings and updating the page table, changes to the page directory are necessary to keep information consistent. In return, this directory allows saving more entries than the page table alone.

In either case, memory accesses might trigger a TLB miss or even page fault. It is the task of the TLB miss handler to perform a lookup in the page table to fill in the missing TLB contents and to transfer control to the DSI/ISI exception handlers in case no page table entry exists. Latter exception handlers may then fetch the mapping information from the page directory and insert the data into the page table, possibly evicting existing mapping information.

With the separation of user address spaces in effect, another source for unwanted information flow is through memory resources for partitions residing in kernel memory. Following the idea of a separation kernel, the PikeOS implementation keeps distinct memory pools for each resource partition. Allocations for kernel data structures (e.g., when creating a new thread) are fulfilled with memory pages from the current partition's pool – subsequent deallocation then returns the page to the originating pool.

In the same manner as with user space allocations, resource partitions cannot be removed, also memory assigned to the kernel page pool of a resource partition cannot be freed and thus reused by partitions other than the originally assigned partition.

For the kernel resource partition page pools, we have shown functional correctness of the memory manager functions using VCC [Bau+11] – together with additional assumptions this implies the desired separation properties for resource partition data structures.

Scheduling and Preemptive Multitasking PikeOS supports preemptive multitasking, i.e., user processes are relinquished of the CPU in favor of other processes. The scheduler of PikeOS processes time partitions according to a user-defined plan that assigns time partitions to time slots with configuration-defined duration. Within a time partition, the scheduler does not work to the effect to establish a fairness property, as threads are scheduled based on priority in a FIFO order – to guarantee that a specific thread is scheduled without relying on a higher priority thread yielding the CPU, a separate time partition may be used.

Besides preemption in user space, a thread may also be suspended when executing in the kernel. For one, an asynchronous interrupt may demand execution time on the CPU. The first levels of the interrupt handling mechanism are fairly short and the real work is done by user space threads that are attached to the interrupt; these threads are not scheduled immediately, but only woken up. This allows the interrupted kernel thread to resume execution quickly. Another possibility for a thread to be suspended in kernel mode is preemption by another thread. Adding preemption in the kernel allows other threads to quickly respond to certain events, as, e.g., in the case of IPC.

However, to avoid race conditions and guarantee consistency of the task's state, transferring control to other threads has to be performed in a controlled fashion. For this, in PikeOS two mechanisms are used that act like a lock on certain data structures: (a) maskable asynchronous interrupts are masked (notably also the decrementer interrupt used to periodically invoke the kernel) (b) preemption by the PikeOS scheduler is disabled. While the interrupt lock, once set, is enforced by the hardware, disabling preemption is a mechanism completely accomplished in software. Correctness of the preemption locking mechanism thus presumes that a locking policy is obeyed by the kernel implementation – this is part of the verification conditions generated from the kernel specification given in Chapter 3. In our single-CPU setup disabling interrupts effectively also prevents preemption by the kernel scheduler through other threads, this is no longer true for multicore systems.

2.4. Deductive Verification

There are many ways to improve the quality of software by eliminating errors in a program, ranging from various testing techniques to model checking and deductive verification. Within the field of deductive verification, available tools not only differ in their target and specification language, but also in the type of required interaction with the user – from mostly interactive tools like the Isabelle/HOL framework [NWP02], to “automatic” tools like VCC [Coh+09b].

In this thesis we focus on the program verification tool VCC, as it was used to verify our case study, the PikeOS microkernel. In the following, we will give an overview of the verification methodology and specification language as far as necessary for the simulation proof presented in Chapter 3 and various examples given in later sections. As a comparison to VCC, which represents the class of tools following the annotation-based paradigm, we will take a look at the KeY system [BHS07]. In contrast to VCC, KeY enables the user to interact with the verification system during the proof search. This allows us to compare the different interaction paradigms and is the basis for further improvements of both annotation-based systems and auto-active tools. In addition, soundness and completeness issues for the axiomatization respectively the taclet base of both tools are discussed and examined in a case study to reveal errors in the axiomatization, respectively rule base in Chapter 6.

2.4.1. The VCC System

VCC is a deductive verification tool for concurrent C programs at the source code level that is used to prove correctness of C implementations against their functional specification. The tool has been developed as part of the Verisoft XT project by Microsoft Research and the European Microsoft Innovation Center (EMIC) to be used for the verification of system software in the hypervisor and avionics sub-projects. The VCC toolchain allows for modular verification of programs using function contracts and invariants over data structures. Function contracts are specified by pre- and postconditions. VCC is an annotation-based system, i.e., contracts and invariants are stored as annotations within the source code in a way that is transparent to the regular, non-verifying compiler.

In the following sections, we give a short introduction to the VCC specification language and the design of the VCC tool. In particular, we will focus on the subset of specification features needed for the examples in later sections. For a general, more thorough primer into using VCC, we refer to the VCC tutorial [Coh+12].

Architecture and Tool Support

As most annotation-based verification systems today, VCC works using an internal two-stage process. The reason for this division is a better separation of concerns and easy integration of different tools. As shown in Figure 2.4, the first stage of the VCC toolchain translates the annotated C code into first-order logic via an intermediate language called Boogie [DL05]. Boogie is a simple imperative language with embedded assertions.

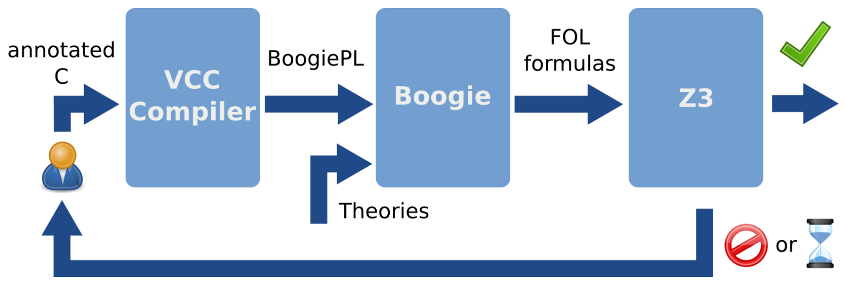


Figure 2.4.: The VCC toolchain architecture

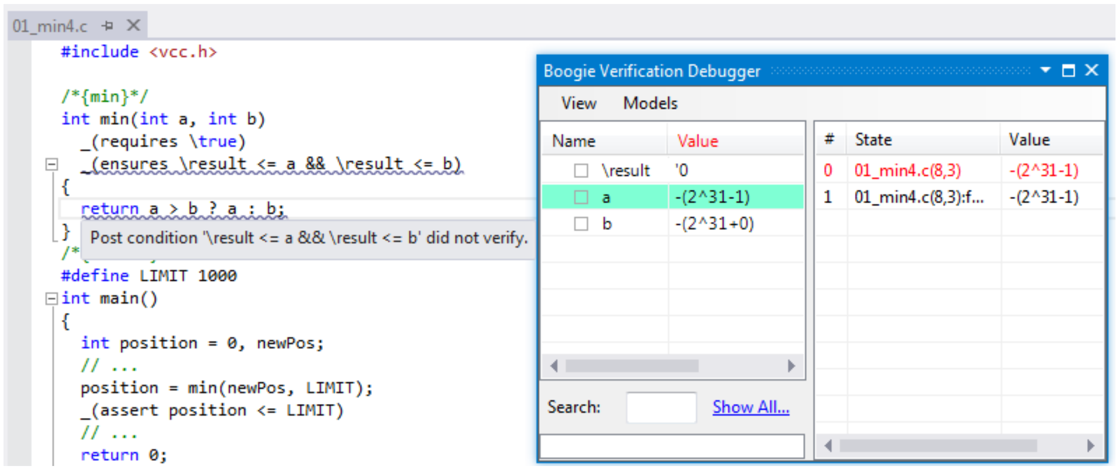
From this Boogie representation, it is easy to generate a set of first-order logic formulas, which state that the program satisfies the assertions. These formulas are called verification conditions and the stage a verification condition generator (VCG).

In the second stage, the resulting formulas are sent to an automatic theorem prover, respectively SMT solver (in our case Z3 [MB08]) together with a background theory capturing the semantics of C's built-in operators, etc. The prover checks whether the verification conditions are entailed by the background theory. Entailment implies that the original program is correct w.r.t. its specification. Interaction with the VCC tool is only possible (and necessary) before the first stage of the toolchain, by providing annotations. Once sufficient annotations have been provided (assuming the program fulfills its specification), the proof is done automatically, hence the term *auto-active* was coined [LM10] for systems following this interaction paradigm.

Other tools following the annotation-based paradigm include Spec# [BLS05] or Frama-C [Cuo+12a]. They are all based on powerful fully-automatic provers and decision procedures, and they support real-world programming languages such as C and C#.

As interaction with the verification tool is restricted to supplying the right set of annotations, good feedback of the tool is important. In this regard, VCC offers integration into a development environment (Visual Studio), e.g., by providing shortcuts to invoke VCC for selected parts of the program and reporting errors in verification using squiggly red lines as part of the main code editor. This error report alone already allows to pinpoint many issues with either the specification or the program, as the exact failing annotation is highlighted for inspection of the user. In addition, the underlying SMT solver used by Z3 is able to provide counterexamples which can then be translated back to sequences of (partial) program states showing relevant information to be able to identify the reason for verification failure. The VCC model viewer (see Fig. 2.5) is a tool to inspect the failing program trace and allows the user to choose various states in the program execution (including specification state).

Another outcome of the verification attempt is that the execution of the tool exceeds a user-defined timeout. Inspecting and improving on long-running verification attempts is possible with the Z3 Inspector tool – during execution of the SMT solver, this tool shows the user which annotation is being currently worked on together with the amount of work spent so far for particular annotations. Further statistics, like the number of quantifier instantiations, allow users to identify possible performance issues and optimize annotations to speed-up verification (e.g., by choosing better triggers for quantifiers).



```
#include <vcc.h>

/*{min}*/
int min(int a, int b)
  _(requires \true)
  _(ensures \result <= a && \result <= b)
{
  return a > b ? a : b;
}
/*
#define LIMIT 1000
int main()
{
  int position = 0, newPos;
  // ...
  position = min(newPos, LIMIT);
  _(assert position <= LIMIT)
  // ...
  return 0;
```

Post condition '\result <= a && \result <= b' did not verify.

Name	Value
\result	'0
a	-(2^31-1)
b	-(2^31+0)

#	State	Value
0	01_min4.c(8,3)	-(2^31-1)
1	01_min4.c(8,3):f...	-(2^31-1)

Figure 2.5.: VCC integration into the Visual Studio IDE and Model Viewer

Verification of Sequential Programs

In the sequential setting, specification of C programs using VCC is done as customary in other annotation-based program verification tools, e.g., those using the Java Modeling Language (JML) to specify Java programs: Functional correctness of a method is specified with function preconditions and postconditions, together with method framing. We distinguish between partial and total correctness – the latter states that if the method is run in a program state satisfying the precondition of the method, then the method terminates (compared to partial correctness which does not require termination) and the postcondition holds after the method body has been executed.

VCC specification language constructs may use side effect free C expressions, together with universal and existential quantification to express predicate logic formulas, as well as special specification keywords (such as `old` to refer to the value of an expression in the pre-state of a method, or `result` to refer to the return value of a method).

As an example for a simple specification, Listing 2.6 shows the contract of a function that returns the smallest element of an array. The precondition "requires" parameter `len` to be greater than zero and pointer `a` to point to an array of size `len` in memory, accessible to the currently executing thread (called "thread local"). The postcondition "ensures" that the result is indeed the minimal element of the array, i.e., (a) it is less or equal than any element in the array and (b) equal to one of the elements of the array.

Method contracts are not only used to lay down the requirement specification of the implementation of a method, but also to modularize the verification task: calls to a method are replaced by its contract, which provides a possibly weaker and hopefully more abstract and succinct representation of the method's effect. While using contracts instead of the actual implementation improves performance of the verification system (e.g., in cases where the method body is long), the absence of the option to inline method calls and the necessity to provide contracts for all methods can be disadvantageous – this issue and possible improvements are subject of discussion in Chapter 7.

```
int min(int *a, unsigned int len)
_(requires len > 0)
_(requires \thread_local_array(a, len))
_(ensures \forall unsigned int i; i<len ==> \result <= a[i])
_(ensures \exists unsigned int i; i<len && \result == a[i])
{ ... }
```

Listing 2.6: Example VCC function contract

Reasoning about loops in the program is done via loop invariants, resulting in the usual verification conditions, i.e., (a) the loop invariant holds before entering the loop body, (b) loop body execution preserves the invariant and (c) as the use case, the invariant is assumed to hold after exiting the loop (so the loop condition is known to evaluate to false). The postcondition of the method has to be established using information from (c), together with the rest of the method body after the loop.

The method of Listing 2.6 is a so-called "pure" function, i.e., it does not change the content of memory locations visible outside the function's scope. Handling change of program state in an imperative language like C efficiently is one of the key issues for program verification tools. While the pre- and postcondition of a function are used to describe precisely how the parts of state the user is interested in have changed due to execution of the function, keeping track of the fact that most of the memory has not been altered is equally important. The latter problem is known as method framing and various approaches have been devised to be able to succinctly express which parts of the state are kept unchanged by execution of a program. With VCC, method contracts can be annotated with a `writes` clause to specify which parts of memory may be modified by the method – this allows for calling methods to infer which parts of the state are unchanged after the method call.

Keeping track of information about the state (either specified by the user or inferred by the verification tool), despite modifications made by execution of the program, is not only simplified using framing, but also by exploiting disjointness properties between data structures derived from ownership relations.

VCC Memory Model One of the properties attributed to the C programming language is that it is "low-level", allowing fine-grained manipulation of data structures at the most basic layer, i.e., as their representation in memory as sequences of bytes. While most of the time this low-level access to data is avoidable, some operations in system programming depend on it. Hence, a verification system has to include measures to handle byte-level access to structured data.

With C's notion of byte-addressable memory, a straight-forward concept of a specification language could map higher level language constructs like structs to sequences of accesses to bytes in linear memory. In early versions of the VCC tool this notion was discernible by the possibility to use memory regions in specifications (e.g., when requiring disjointness of data structures). However, performance issues with this region-based specification approach, as mentioned by Alkassar et al. [Alk+10c], required a different strategy of handling memory. As interaction with structured data on the byte-level is not the common case, but most of the time, C structures are accessed and modified using field references, this suggests treating such structures as proper objects (including a mechanism to support the infrequent byte-level access).

As a consequence, pointers to memory are associated with a type which helps to simplify deducing aliasing properties – in this pointer model, e.g., pointers of different primitive types never alias. A detailed account on the currently used heap model in VCC and on the history of the different heap models used by older VCC versions is given by Böhme and Moskal [BM11]. If a memory region is to be used for different purposes, i.e., to store data with varying types (e.g., as in the verification of the PikeOS kernel memory allocator [Bau+11]), memory has to be reinterpreted and the VCC specification language provides special annotations to accomplish this.

Ownership and Object Invariants Access (both reading and writing) to data structures is governed in the VCC methodology by the rules of *ownership*. An object can be in different ownership states throughout the execution of a program: In the sequential case (i.e., no variable is marked with the C keyword volatile and we assume no other threads are running), an object is either owned by the (only) running thread (referred to with the keyword `me`) or by another object. Thus, in general, the ownership relation forms a tree with a thread as root. In addition, an object can either be closed or open. Modifications to fields are only possible if (a) the current thread is the immediate owner of the object and (b) it is open.

Closed objects directly owned by the current thread are called `wrapped` in the context of this thread. Ownership and object state can also be manipulated and transferred. In the simple case, an object is passed `wrapped` into a method which then unwraps the object, performs updates on its fields and wraps it again before returning.

Another feature in VCC are *invariants* on objects, specifying properties a "consistent" state is supposed to abide to. Compared to loop invariants, which capture properties about the state at determined points in the execution (i.e., just before the loop execution and after each loop iteration), in case of object invariants, the user is responsible to specify when an object is considered to be in a valid state and its invariant should hold. For VCC, invariants are defined to hold for objects that are closed – so wrapping an object includes the verification condition that the object's invariant holds in the current state. Invariants are not allowed to refer to arbitrary memory locations (which would make checking them difficult) but only depend on fields of objects in the ownership domain of the object the invariant belongs to. As wrapping an object requires all children (i.e., owned objects) to be wrapped, their invariants are known to hold and may be used to infer the invariant of their parent – changes to the children are not possible after wrapping their owner and thus the parent's invariant cannot be invalidated as long as it is closed.

Specification State Besides the ordinary memory available to the implementation, specifications can make use of ghost (or specification) state and specification code. Ghost state is disjoint from the program-accessible memory and is not allowed to be accessed by C statements, but only by specification statements. This guarantees (together with termination of specification code) that program execution is not altered by specification statements added to a C program.

Use cases of specification state are to serve as a means of abstraction (e.g., representing the elements of a linked-list data structure as a set of objects), making use of abstract data types like unbounded maps and integers, or to record intermediate computation results needed later on in an annotation (e.g., a witness for an existential quantifier). Specification state may be added in form of additional fields of a structure and is then usually coupled to implementation state with invariants; ghost variables may also be passed directly as *ghost parameters* to methods. Ghost code is also used to define pure methods that can then be used in specifications, or to prove statements with an inductive-style proof making use of while loops together with invariants.

Specification state and statements have to be added by the user of the verification tool and are often a prominent example for a kind of annotation we call *auxiliary*, in contrast to the requirement specification which serves as the definition of the expected behavior of a program. In Chapter 5, we will show that the definition of auxiliary annotations can further be subdivided according to their role in the verification task into essential annotations (e.g., loop invariants) and non-essential annotations.

Verification of Concurrent Programs

Modular verification of concurrent C programs using VCC is achieved by specifying the changes allowed to concurrently (i.e., `volatile`) modifiable state with so-called *two-state invariants*. The verification methodology of VCC ensures that each thread abides to the protocol given by the two-state invariants, allowing, e.g., to verify programs requiring cooperation for correct operation.

As an example for two-state invariants, consider the following simple C structure modeling a clock with the requirement that the value of field `time` can only be increased:

```
struct clock_str {
    volatile int time;

    _(invariant time >= 0)
    _(invariant \unchanged(time)
        || time > \old(time))
}
```

Both two-state and "one-state" object invariants may be combined – in this example, a one-state invariant further restricts the values of field `time` to those greater than zero, while the two-state invariant enforces that `time` either stays unchanged (`\unchanged(x)` is an abbreviation for `\old(x) == x`), as each two-state invariant has to allow stuttering, or is increased.

For volatile data, there are two ways to modify it: As for the sequential case, volatile data can be read and written by a thread owning the corresponding object exclusively (by unwrapping it first). In this case, when unwrapping an object with a volatile field, the two-state invariant is irrelevant when modifying this field. In addition, volatile fields of *closed* objects can also be written to. In order to guarantee consistent updates to such an object, changes to volatile fields in the latter case have to be enclosed in an *atomic* block, as follows:

```
struct clock_str clock;
clock.time = 1;
_(wrap &clock)

_(atomic(clock)) {
    inc(clock.time)
}
```

After initializing the structure `clock` to satisfy the invariant of `clock_str`, we are able to wrap the data structure. As the name implies, in an atomic block only a single, atomic access (read or write) to memory can be performed – changes to the volatile field have to fulfill the two-state invariant of the enclosing structure. To keep this example as simple as possible, the method call `inc` is supposed to correspond to an atomically executed machine instruction. In a more realistic example this could be replaced by a lock-free algorithm using a compare and swap (CAS) instruction more commonly found on real hardware. Besides the single access to physical memory, arbitrary many read and write operations can be performed on ghost state, e.g., to satisfy coupling invariants.

Another prerequisite for modifying volatile state of closed objects is that the currently executing thread has the guarantee that the object stays closed when updating it within an atomic block – if another thread could open the object in the meantime, maintaining the two-state invariant would be infeasible. In our example this knowledge comes from the fact that structure `clock` is an automatic C variable and private to the enclosing block.

If an object like `clock` was used in a real setting, it would be shared by multiple threads, and we would have to ensure that all threads cooperate by performing only updates to volatile state according to the two-state invariant. Therefore, in terms of the ownership discipline, no thread may obtain exclusive access to the shared object (which would otherwise allow this thread to unwrap the object and change it arbitrarily).

Instead, the assurance that the shared object stays closed and that user-provided invariants about the (volatile) state of the object hold is established by using specification objects called *claims* [Dah+09]. Claims are created (and destroyed) using ghost statements. When creating a claim on an object o, the user may provide an invariant I for o – a proof obligation is generated to verify that I holds in the current state and that I is maintained as long as the referenced object stays closed (i.e., volatile state updates obeying the two-state invariant of the object do not falsify I). The verification methodology of VCC ensures that objects with outstanding claims attached cannot be opened, thus holding a claim on an object guarantees the invariant of the claim. To be able to unwrap an object, first all claims on the object have to be destroyed.

As already mentioned, modifying volatile state of an object in the VCC methodology requires the knowledge that the object stays closed – if this knowledge is established using claims, these claims have to be given as parameters to the call to atomic(). For our example above, the only object that needs to be included in this call is the structure clock itself.

Changes to volatile state by other threads only happen before entering an atomic block, in effect restricting scheduling of the threads in the system. That this coarse scheduling is sound is guaranteed by the order reduction theorem sketched by Cohen et al. [Coh+09c].

In the previous example, one guarantee that can be derived via the claim mechanism is that reading field time twice in succession, the second value will be greater or equal to the first one. For non-exclusive access to structure clock, this is put into a claim as follows:

```
void clockClaim(struct clock_str *clock _(claim c))
    _(requires \claims(c, clock->\closed))
{
    int currTime;
    _(atomic(clock, c)) {
        currTime = clock.time
    }

    _(claim cPrime = claim(c, clock.time >= currTime))
}
```

As data structure clock is shared with other threads, a claim c has to be passed to the method ensuring that clock stays closed. After reading the current value of the time field, a new claim can be created, stating that field time is greater or equal to the last value read. This claim is based on the old claim c, as the newly claimed property relies on the fact that all updates to field time respect the two-state invariant of structure clock.

In a more realistic setting, VCC has been used to, e.g., specify and verify locking mechanisms using volatile state together with claims to manage transfer of ownership when acquiring and releasing a lock [HL09]. Also, lock-free algorithms have been verified using VCC's verification methodology.

2.4.2. The KeY System

The KeY[2] tool [BHS07] is a verification system for sequential Java (Card) programs. Similar to VCC, programs can be specified using annotations in the source code. In KeY, the Java Modeling Language (JML) is used to specify properties about Java programs with the common specification constructs like pre- and postconditions for methods and

[2]See http://www.key-project.org

object invariants. In addition to VCC's ghost state and programs, KeY also features model fields and model methods that can be used similarly. Like in other deductive verification tools, the verification task is modularized by proving one Java method at a time.

For comparison, the contract in VCC for the method shown in listing 2.6 is translated to JML as follows:

```
/*@
  @  requires a != null && a.length > 0;
  @  ensures (\forall int i; i >= 0
  @               && i < a.length; \result <= a[i]) &&
  @          (\exists int i; i >= 0 && i < a.length
  @               && \result == a[i]);
  @  assignable \nothing;
  @*/
public int min(int[] a)
{ ... }
```

For this simple example, both the VCC and JML version of the contract are quite similar. Besides small changes in syntax, the only change in the contract is the property that array `a` is non-null instead of the requirement in VCC that parameter `a` points to allocated memory owned by the thread (annotation `\thread_local_array(a, len)`).

In the following, we will briefly describe the workflow of the KeY system. For this example, we assume the user has chosen one method to be verified against a single pre-/postcondition pair. First, the relevant parts of the Java program, together with its JML annotations are translated into a *sequent* in Java Dynamic Logic, a multi-modal predicate logic. Total correctness of a program p wrt. the precondition pre and postcondition $post$ is, in a simplified version, expressed by the DL-formula $pre \rightarrow \langle p \rangle post$. For our previous example, the translation of the pre-/postcondition pair results in the JavaDL sequent shown in Listing 2.7 (again simplified for readability).

Here, `ValidState` abbreviates a formula stating certain well-formedness properties of the heap and other conditions a valid state is supposed to adhere to. As seen in this example, there is a close correspondence between the JML-annotations and the resulting DL-formula: the precondition of the method is represented by (parts of) the antecedent of the implication, the program to be proven correct is contained within the diamond modality (enclosed by statements to be able to refer to exceptions thrown by the method) and the postcondition is contained in the formula directly after the modality. Correct framing is proven by discharging the last conjunct of this formula.

Validity of this sequent implies that the program is correct w.r.t. its specification. Proving the validity is done using automatic proof strategies within KeY which apply sequent calculus rules implemented as so-called *taclets*, together with user interaction, e.g., in form of manual taclet application to instantiate quantifiers or to introduce lemmas. In addition, KeY provides an interface to a number of SMT solvers which the user may invoke and which are suitable for some proof goals where validity of the formula can be deduced by examining parts of the sequent that falls into supported theories of the SMT solver chosen.

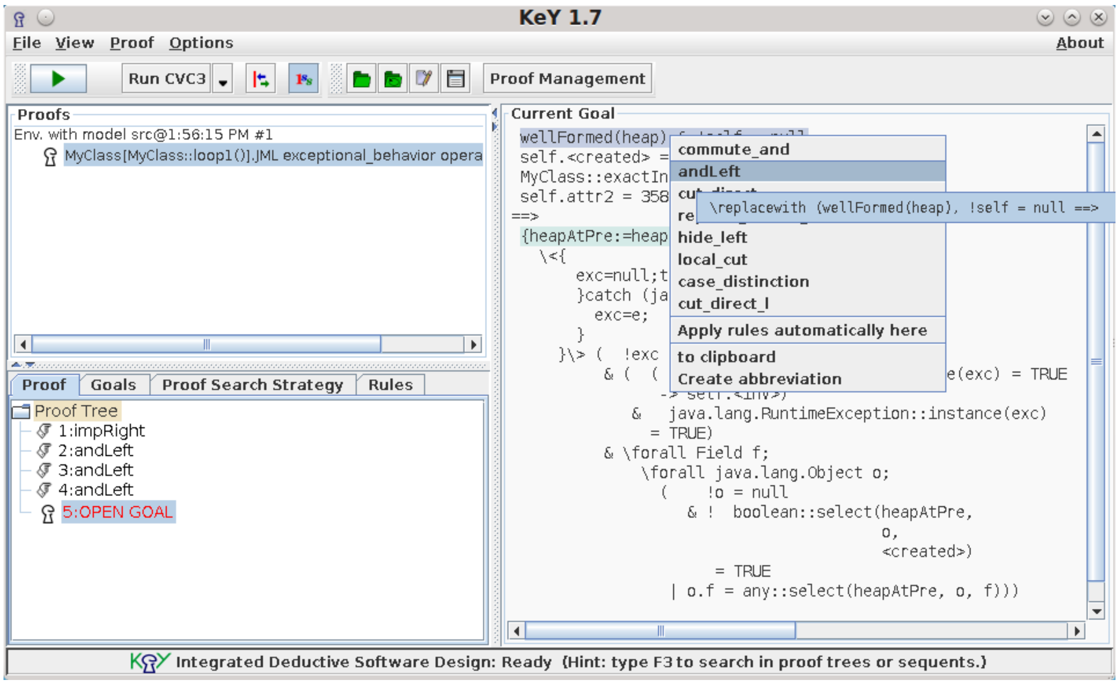
```
==>
ValidState ∧ (¬a = null ∧ a.length >  0 ∧ (self.<inv> ∧ ¬a = null))
 → {heapAtPre:=heap || _a:=a}
   \<{
       exc=null;try {result=self.min(_a)@Min;
       }catch (java.lang.Throwable e) { exc=e; }
     }\>
     (∀ int i; (i ≥ 0 ∧ i < a.length ∧ inInt(i) → result ≤ a[i])
      ∧ ∃ int i; (inInt(i) ∧ (i ≥ 0 ∧ (i < a.length ∧ result = a[i])))
      ∧ self.<inv> ∧ exc = null
      ∧ ∀ Field f; ∀ java.lang.Object o;
         (¬o = null ∧ ¬heapAtPre[o.<created>] = TRUE
            ∨ o.f = heapAtPre[o.f]))
```

Listing 2.7: JavaDL proof obligation for method `min`.

The set of taclets provided with KeY plays a similar role as the prelude in case of VCC, as it captures the semantics of Java, built-in abstract data types like sequences etc. In comparison to the prelude of VCC, however, KeY also contains taclets that deal with first order logic formulas, whereas first-order reasoning in VCC is handled by the SMT component Z3. Programs inside the modalities of a JavaDL formula are handled by symbolic execution. These rule applications dealing with Java statements and expressions can be performed fully automatic (modulo required user-supplied JML annotations for loops and methods depending on strategy settings). Compared to VCC, KeY offers some choice in handling loops and methods: loops can be unrolled (suitable for only a small number of known loop iterations) or reasoned about using induction, and method calls can be replaced by their bodies (suitable mostly for smaller methods, due to performance reasons).

Results of a verification attempt in KeY are also similar to those in VCC: either the generated Java Card DL formula is valid and KeY is able to prove it; or the generated formula is not valid and the proof cannot be closed; or KeY runs out of resources.

In contrast to VCC, automatic proof search of KeY can be combined with interactive steps of the user, in case a proof is not found automatically. Figure 2.8 shows the user interface of KeY with the first step of a rule application completed. In the main window the sequent of a currently open goal is shown. The user is then able to select any term or formula in this sequent and is shown a list of *applicable* taclets, in this case eight taclets are available for the currently selected term. After manually applying one or more taclets (e.g., a manual quantifier instantiation), automatic proof search can be resumed. Compared to VCC, locating the exact JML annotation that cannot be proven by KeY to hold is not as convenient, as the user has to inspect which proof branches are left open (where one branch often does not directly correspond to a single annotation). Starting with KeY version 2.0.0, this situation has improved by being able to change the proof search strategy to split the proof according to the different postconditions that have to be shown.



```
 wellFormed(heap)
 self.<created> =
 MyClass::exactIn
 self.attr2 = 358
==>
 {heapAtPre:=heap
   \<{
      exc=null;t
      }catch (ja
        exc=e;
      }
    }\> (  !exc
        & (  (                          e(exc) = TRUE

           &   java.lang.RuntimeException::instance(exc)
             = TRUE)
        & \forall Field f;
            \forall java.lang.Object o;
              (     !o = null
                 & !  boolean::select(heapAtPre,
                                      o,
                                      <created>)
                    = TRUE
               | o.f = any::select(heapAtPre, o, f)))
```

```
\replacewith (wellFormed(heap), !self = null ==>
```

Figure 2.8.: Graphical user interface of the KeY verification system

3. Verifying a Paravirtualizing Microkernel

Contents

Demonstrating the correctness of a virtualization kernel involves different aspects: (a) the high-level, informal notion of what constitutes correct operation of a virtualization kernel and, as an instance of this, the desired properties of the concrete system running the kernel implementation, (b) a formal specification on a suitable abstraction level to be able to ensure these high-level properties, together with a definition of correctness of an implementation w.r.t. a specification and (c) the actual proof that the concrete implementation of the virtualization kernel is correct w.r.t. the given formal specification.

Concerning the high-level correctness properties of the system (a), we have given an introduction to the intended functionality of system virtualization in Section 2.1 (e.g., emulating the host hardware, at the same time separating guest systems in full virtualization). In this chapter we will give a definition of correctness of a virtualizing kernel in terms of a simulation property, addressing item (b) above, as well as demonstrating how to use the VCC verification methodology to conduct the corresponding simulation proof for PikeOS, as part of item (c).

In Section 3.1, we start with a brief general introduction to specification of reactive systems on the level of abstract state transition systems, together with simulation properties (Sec. 3.1.1) and present the concrete simulation property used for the verification of PikeOS (Sec. 3.1.2).

The main part of the chapter is concerned with proving this simulation property with the help of VCC (Sec. 3.2). We will start by showing how to encode the simulation property as VCC annotations; in the first step, for a simple hypervisor, reporting on the work by Alkassar et al. [Alk+10c] in Section 3.2.1.

We continue with applying VCC to verify a code sample in the form of a simple PikeOS system call in Section 3.2.2 in the sequential setting. This system call example already demonstrates some issues in using auto-active verification tools as discussed in Chapter 4. As explained in Section 2.3, PikeOS is a preemptible system. Of course, execution of the kernel cannot be preempted at arbitrary points in time, as certain parts of the kernel implementation expect to be run sequentially as they require a consistent kernel state. Those sequentially executed portions of the kernel may be specified and verified just by method (or block) contracts and thus were an expedient starting point for our early verification efforts within the avionics subproject.

With these preliminaries covered, we then present the general setup for the verification of PikeOS in the *concurrent* setting in Section 3.2.3, mainly using the system call from Section 3.2.2 as example. For this simulation proof, instead of reasoning over behavior of complete runs of the system, we ensure that all single transitions relevant for the virtualized guest systems are correct. For this, VCC provides the mechanism of ownership and two-state invariants to restrict (concurrent) updates to the state to permissible ones. We also demonstrate how locking mechanisms may be used to capture the effect of restricting preemptability of the kernel via disabling interrupts, respectively prohibiting preemption by the kernel scheduler.

Concerning the results of the avionics subproject within Verisoft XT, due to limited resources, we were not able to conduct a full functional correctness proof of the whole PikeOS kernel – instead, we restricted our attention to parts of the kernel that implement core mechanisms relevant to establish safe and secure virtualization (e.g., correct operation of the memory manager [Bau+11]). One contribution concerning the PikeOS verification task in Verisoft XT is the adaptation of the specification methodology developed in the Hypervisor subproject to the preemptible kernel PikeOS. We claim that this methodology, presented in the following sections, is suitable to be extended to the remaining parts of the PikeOS implementation and, more generally, to other preemptible microkernel variants with a similar setup (e.g., systems with single processor).

Also, in this thesis, we only provide a high level overview of this general verification methodology for preemptible kernels with VCC – many aspects that are necessary for the correct operation of the virtualization feature are out of scope of this thesis and are either assumed to hold (e.g., soundness of VCC or correct translation of C code to machine code by the C compiler) or presented in other publications – we briefly give an overview of some remaining parts of the correctness argument in Section 3.3. Finally, we conclude this chapter with a brief summary of two related system software verification projects in Section 3.4.

In our verification methodology, we draw heavily upon the results obtained by the developers of the VCC verification tool, as well as the team responsible for Hyper-V verification – for the VCC specification examples presented further we especially would like to acknowledge support by Sabine Schmaltz and Ernie Cohen.

3.1. Correctness of Virtualization Kernels as Simulation Property

Compared to functional correctness for sequential programs, where we may describe behavior of the program as a function from input to output, reactive systems like operating systems are more amenable to specification in terms of an abstract transition system. To verify that the implementation is correct w.r.t. this specification as transition system, we have to demonstrate that the observable behavior of the implementation is similar to that of the transition system.

The exact notions of similarity and observability to be used depend both on the kind of properties we would like to show for the concrete implementation and the capabilities of the observer of the concrete system. For example, for our virtualization setup, an observer that is able to inspect the whole CPU state is too powerful, as separation properties of the kernel build upon the fact that processes have restricted visibility of the state established by mechanisms of the hardware (e.g., the system mode, restricting access to supervisor-level registers). One of the various possible descriptions of an observer could be one that is only able to observe the interaction between the concrete system as a "black box" and its environment.

The specification of the concrete system by means of an abstract transition system allows analyzing whether the system works as intended and to derive high-level properties that are more approachable to a common understanding (e.g., concerning safety or information flow properties). For our purposes of verifying safety properties of the abstract system, the implementation of the concrete system is not allowed to exhibit more observable behaviors than specified by the abstract transition system – otherwise, even if on the abstract level all states of the system have a desired property P, the concrete system could have additional behavior and thus states for which P does not hold. As the abstract transition system is allowed to have states (and transitions) not present in the concrete system, any property relying on the existence of a certain execution trace or state cannot be transferred from the abstract down to the concrete level with our definition of similarity.

Verification of similar behavior of both abstract and concrete system is achieved by simulation proofs, as described in the following.

3.1.1. Simulations

Various different simulation proof techniques are defined in the literature that may be used to verify correctness of concurrent systems. A good overview of the different variants is given by Lynch and Vaandrager [LV95], also Glabbeek [Gla01] provides a survey of semantic equivalence concepts found in the literature.

For our purpose of showing correct operation of virtualization kernels by proving that concrete execution simulates the abstract one, we are interested in the property that each computation on the concrete level also is a valid computation on the abstract level, tantamount to trace inclusion: According to Glabbeek [Gla01], the coarsest preorder between states of two transition systems is induced by the notion of *trace semantics*.

In trace semantics, given two transition systems, an abstract system A and a concrete system B, if all *traces* (described as sequences of external events) of the concrete system are also traces of the abstract system, then B implements A. Often, simulation of one system by another is defined modulo stuttering of the abstract system, i.e., the transition relation is reflexive and repeated occurrence of the same state in a trace is permitted.

Reasoning on traces of the different systems is usually reduced to proving properties of single transitions of the systems, e.g., by using different simulation techniques that imply this trace inclusion property, like refinement mappings [AL91], forward simulation [Mil71], or the stronger notion of bisimulation [Mil83].

As we are only concerned about the fact that the *observable* behavior of the concrete system can be matched by the abstract system, there has to be a notion about what is observable (to the environment). One option is to define traces of a transition system as a sequence of labels – transitions have either labels corresponding to observable interactions with the environment or a special label indicating a *silent transition* (cf. Jonsson [Jon91] or Lynch and Vaandrager [LV95]). Another option is to use a state-based definition of simulation, e.g., as described by Abadi and Lamport [AL91] for refinement mappings, where states are divided into an internal and observable part. In both cases, the simulating system is only required to perform equivalently up to the externally visible part of the trace of the system.

In the following, we will use the technique of forward simulation to prove equivalent behavior of the PikeOS implementation and the abstract system specification. According to, e.g., Jonsson [Jon91], (forward) simulation goes back to Milner [Mil71]. Forward simulation is defined as a relation R between states of the concrete and abstract transition system, s.t. (a) each concrete initial state has a corresponding abstract initial state (given by relation R) and (b) each transition on the concrete level from a state coupled to an abstract state via R has a counterpart on the abstract transition system. The existence of such a forward simulation relation then implies trace inclusion.

In transition-based simulation approaches, the equivalence of system steps for the different systems is given by the abstraction of events and proving the existence of such a relation R suffices to show similar behavior of the two systems. In case of state-based simulation definitions, an adequate definition of the simulation relation R is crucial to be able to infer any meaningful property about the concrete system. This fact is, e.g., also explicitly mentioned by Leinenbach [Lei08] in the context of his work on a compiler correctness proof using a simulation argument within the first Verisoft project, using the trivial universal relation as an extreme example for an inadequate simulation relation.

3.1.2. The PikeOS Simulation Theorem

Correctness of the PikeOS implementation is defined in terms of the behavior of an abstract model of the overall system. For the verification of PikeOS, the particular model we used within Verisoft XT was based on the computational model named Communicating Virtual Machines (CVM) [Gar+05], as established in the first Verisoft project. The idea behind CVM is to model user processes and hardware devices interacting with an *abstract* kernel – the low-level hardware operations of a concrete kernel containing assembly are performed by so-called kernel primitives. To get a concrete kernel, the missing kernel primitives have to be implemented and are linked against the abstract kernel. Within the first Verisoft project, two concrete kernel implementations (VAMOS [Dör10] and OLOS [DSS09]) were successfully combined with the CVM model.

However, for the CVM model, the kernel was assumed to be non-preemptible, i.e., each of the primitives implemented by a concrete kernel results in one step of the abstract system. In addition, memory of user processes was separated, thus communication was only possible via defined primitives, unlike for PikeOS processes, where memory sharing is allowed. As a consequence, the CVM model had to be extended for our purposes. In the following, we will present parts of the updated CVM model used for modeling the abstract system due to Baumann et al. [Bau+10]. For an introduction to CVM we refer to the work of Gargano et al. [Gar+05], a formalization of CVM together with the corresponding correctness theorem is given by Tsyban [Tsy09].

Figure 3.1 gives a high level overview of the specification state involved in the verification of PikeOS. On the CVM layer, in addition to the user CPU context for each user (i.e., a subset of the registers of the underlying hardware) and the user accessible memory (grouped as *guest systems* in Figure 3.1), the abstract configuration of a kernel thread (in case the user process enters system mode, e.g., via system calls or due to interrupts) is part of state cvm. The concrete state, further denoted by h, consists of the register set of the physical machine, as well as the physical machine memory (depicted as the bottom layer called "hardware model" of Figure 3.1).

For both the abstract CVM layer and the implementation layer a transition function is defined as δ_{cvm} and δ_h respectively. In this formalization, both the abstract model and the implementation are deterministic in their behavior. Possible non-determinism in the abstract model as, e.g., the next user process to be scheduled by the kernel, or the occurrence of external interrupts is fixed by including the scheduler in the abstract CVM model, respectively providing a list of interrupts as parameter of the transition function (not shown here to simplify presentation of the function signatures). The simulation correctness theorem accordingly quantifies over all possible schedulings and interrupt signal inputs to obtain a complete simulation property.

A prerequisite for the simulation proof is a simulation relation between CVM and hardware layer which specifies when an abstract state and a concrete state correspond to each other, i.e., when the behavior of the two systems originating from these states is similar. This simulation relation is further called $cvm\text{-}sim$. In our case, $cvm\text{-}sim$ includes that the states of user processes in the CVM model are properly encoded in the hardware state – i.e., (a) when a process performs a step in user mode, the current CPU state of the abstract user process is stored in the actual registers of the physical machine and (b) information about all user processes not currently executing is stored in data structures in kernel memory (in process control blocks). This part of the simulation relation is called $\mathcal{B}$-relation and was introduced by Gargano et al. [Gar+05].

Besides proper encoding of CVM state in the hardware state, implementation invariants of the concrete state have to hold during system execution, e.g., well-formedness of data structures in the kernel. These invariants are abbreviated by predicate $impl\text{-}inv$.

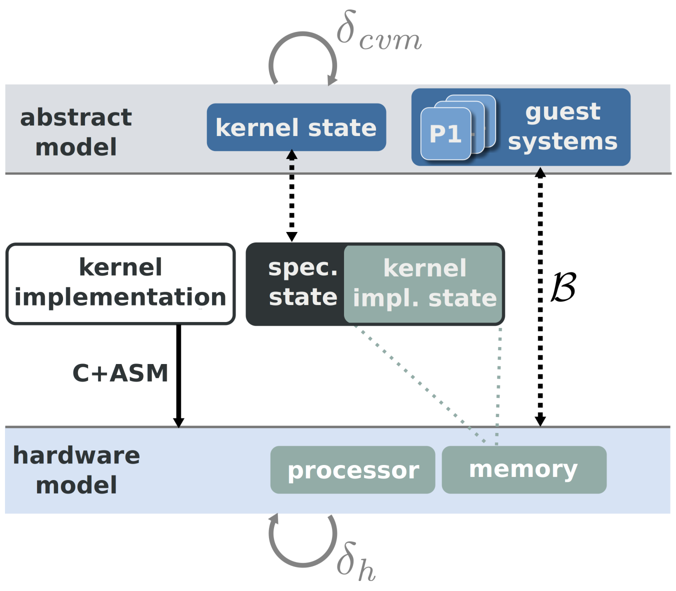


Figure 3.1.: Overview of the specification state and coupling relations for PikeOS verification (simplified version of the illustration by Baumann et al. [Bau+10]). Dashed lines stand for coupling invariants, the solid black line indicates that the kernel implementation operates on the hardware model state. Gray solid lines represent transition relations on the associated model.

In the following, suppose cvm^0 and h^0 are the initial states of the abstract model respectively the concrete system. As part of the correctness theorem for PikeOS, we demand in previous work [Bau+10] that these initial states correspond to each other according to the simulation relation $cvm\text{-}sim$, and for any execution of the abstract system of n steps (of δ_{cvm}) there is a number of steps of the concrete system m (of δ_h), s.t., together with our PikeOS kernel implementation k the resulting states again relate to each other according to $cvm\text{-}sim$, formally:

$$\begin{aligned}&cvm\text{-}sim(cvm^0, k, h^0) \wedge \\ &\forall n\, \exists m \left(impl\text{-}inv(\delta^n_{cvm}(cvm^0), k, \delta^m_h(h^0)) \wedge \right. \\ &\qquad\quad \left. cvm\text{-}sim(\delta^n_{cvm}(cvm^0), k, \delta^m_h(h^0)) \right)\end{aligned} \tag{3.1}$$

The abstract system simulating execution of the concrete system in this definition seems to contradict the notion of correctness as presented in the previous section, if we want to transfer safety properties from the abstract to the concrete system level. However, in this case of deterministic transition systems (as both δ_{cvm} and δ_h are functions), a theorem by Park [Par81] states that the notion of bisimulation and trace semantics actually coincide. This allows us to specify simulation in the more convenient direction from abstract to concrete system in the definition above, i.e., we are able to relate a *single* step of the abstract system to multiple steps of the concrete system.

In the actual verification of the PikeOS kernel with VCC, we adapt this simulation approach to the VCC methodology, which provides specification patterns to deal with concurrent C execution of the preemptible microkernel: Instead of fixing inputs of the environment in form of external interrupts or steps of hardware devices of the system, we specify behavior of both the abstract and concrete system level by transition *relations*.

3.2. Simulation Proofs with VCC

Given the definition of the simulation theorem (3.1) as correctness property of PikeOS in the previous section, the next step is to encode this simulation property in terms of annotations for the VCC tool. This includes the following specification and implementation components:

(a) the abstract model state (cvm in formula 3.1), with the corresponding transition function (or relation) δ_{cvm}, constituting the requirement specification of the overall system,

(b) the state of the host machine (h in Formula 3.1), together with hardware transition function δ_h,

(c) a definition of the predicate $cvm\text{-}sim$, relating corresponding concrete and abstract machine states and

(d) a rendition of the actual simulation property in form of an annotated C program, using the preceding specification components.

In the following, we demonstrate how to implement these four components using the VCC specification language. For the case of a sequentially executing virtualization system (i.e., running on a single CPU with a non-preemptible system software), we will use results of the work on a simple hypervisor (called *baby hypervisor*) by Alkassar et al. [Alk+10c]. The baby hypervisor has been implemented and verified as part of Verisoft XT (in addition to the work on the Hyper-V verification in the Hypervisor subproject), where it “played an important role of driving the development of the VCC technology and applying it to system verification.” [Alk+10c]

The baby hypervisor implementation and specification provides instantiations for the components used in the simulation theorem as described above and gives an overview of conducting the hypervisor simulation proof with VCC. Note that we do not describe the sequential simulation proof using PikeOS, as we directly moved on to concurrent simulation from our first verification results of functional correctness of single system calls in the sequential setting.

We will use the results presented by Alkassar et al. [Alk+10c] as a starting point for the description of our verification methodology for the PikeOS kernel in subsequent sections – first, we recapitulate the specification artifacts and verification technique used for the baby hypervisor correctness proof and continue by extending respectively adapting the specification to suit a preemptible system like PikeOS.

3.2.1. Sequential Systems

Representing both the abstract and concrete system state requires the implementation language (extended by annotations) to provide appropriate constructs suitable for the different abstraction levels – regarding the memory representation of the concrete hardware state in the model, Alkassar et al. [Alk+10c] note that "C memory is sufficiently low-level". Similarly, encoding the CPU state of the modeled host machine (a RISC processor called VAMP [Bey+06]) is conceptually straightforward, as shown by the following (simplified) data type declaration:

```
typedef struct proc_t {
    v_word gpr[32];
    v_word spr[17];
    v_word dpc, pcp;
} proc_t;
```

General and special purpose registers are modeled as arrays of type `v_word` (defined as 32-bit wide integers); the two program counters are represented by two separate fields of the same type. The original data type definition in the source code in the work of Alkassar et al. [Alk+10c] also includes invariants on certain registers, which we omit here. As the baby hypervisor provides full virtualization, the virtualized machine state in this example is identical to the host state and thus the declaration of the data type for the abstract machine state is equal to `proc_t` modulo replacing C arrays by the VCC map specification type.

Transitions of the abstract system are given as specification functions operating on an abstract machine state data structure (combining aforementioned virtualized CPU state and memory) – this transition function, together with the VCC structure modeling the virtualized machine state build the *architecture specification*. Again, due to the similarity of the host and virtualized architecture, the specification functions describing the abstract machine steps can be used in the contract for the C implementation `sim_vamp` which simulates the step function of the host machine.

A first version of encoding the property of the baby hypervisor running on the host machine simulating virtual guest machines, the authors call *basic simulation pattern*, is done by the following loop,[1] modeling execution of the physical hardware:

```
sim_vamp(h, INT_RESET)
while(1)
    _(invariant I && \forall uint i; i < ng ==>
        \exists \natural n; R(abs(\old(h), i), abs(h, i), n))
{
    sim_vamp(h, INT_NONE)
}
```

[1]This example from the work by Alkassar et al. [Alk+10c] has been simplified for brevity and adapted to the current VCC specification language syntax.

Besides the host machine state `h`, function `sim_vamp` takes another parameter that models which external interrupt occurs in the current state of the system – this way, execution of this simple host machine setup is completely deterministic. In this example, the architecture defines only a single external interrupt (reset, represented in the code by the constant `INT_RESET`) which is assumed to only occur once at the beginning of the execution of the physical machine – in all other steps, the value `INT_NONE` indicates the absence of external interrupts. For a model of a more realistic hardware architecture with multiple external interrupt sources, a (underspecified) fixed sequence of interrupt events could be provided as input for the calls to `sim_vamp`.

Similar to an induction principle, the simulation correctness property is given by the loop invariant with the help of predicate `R(`h, h', n`)`, defined as the reachability of state h' starting from state h via step relation of the abstract machine in exactly n steps. The correspondence between abstract and concrete state as stated in the formalization of the simulation relation in Formula 3.1 is given by the abstraction function `abs(`h`)`.

For each of the guest machines, the user has to provide annotations establishing that a single host machine step corresponds to a valid transition on the abstract guest states (including stuttering steps): starting from guest states which are reachable from the initial state just before loop execution (given by the expression `\old(h)`), the updated host state after a single step of the host machine corresponds to an abstract state which can also be reached from the initial abstract state.

Of course, reasoning about single steps of the physical machine in this simulation proof via `sim_vamp` is only appropriate for machine code instructions of the guests – verifying properties about the execution of the baby hypervisor should make use of the fact that most of the implementation is given in a high-level programming language. The authors take this into account by introducing an *extended simulation pattern* in which consecutive calls to `sim_vamp` in the loop execution are grouped in case they correspond to instructions of the baby hypervisor execution – this allows treating these instruction sequences in verification by using VCC function contracts of the hypervisor C implementation.

Before we demonstrate the methodology to verify concurrent execution of system calls in the following, we will first show how the verification of the sequential execution of a system call in PikeOS is accomplished, using (parts of) the specification state introduced in this section for the baby hypervisor verification.

3.2.2. Verifying PikeOS System Calls – Sequential Execution

As we are concerned with specification and verification of the more abstract kernel layer of PikeOS on top of the ASP and PSP components (see Figure 2.3 on page 20), one of the main ingredients for correct kernel operation is the functional correctness of the system calls. We start with a simple full functional specification of a PikeOS system call in the sequential setting using method contracts. While this is an oversimplification compared to the real behavior of the system call, this specification allowed us to get a first intuition of verification using VCC – also, the concurrent verification methodology was not implemented right from the beginning of Verisoft XT. Even this simple setting, however, already introduces models both for representing the PowerPC hardware (similar to the data structure modeling the VAMP architecture described previously) and an abstraction of the kernel data structures, which we will re-use in the subsequent sections.

A Simple PikeOS System Call

The first target we have chosen for system call verification is `p4syscall_fast_set_prio`, which changes the priority of a thread. The single parameter `newprio` of the system call may not exceed the user-configured *Maximum Controlled Priority* (MCP), as stated by the PikeOS kernel documentation:

> "This function sets the current thread's priority to newprio. Invalid or too high priorities are limited to the caller's task MCP. Upon success, a call to this function returns the current thread's priority before setting it to newprio."

The corresponding implementation according to this requirement specification of the system call functionality is straightforward, as shown in Listing 3.2. Function `p4syscall_fast_set_prio` first determines the currently running thread in the system using function `p4_runner` which returns the corresponding kernel data structure used to manage the thread. Afterward, the function changes the actual priority to be set by amending values for parameter `prio` that are out of range to the MCP of the thread in question; performing the change of the thread's priority is then delegated to the helper function `p4_runner_changeprio`. Besides the kernel data structure for the thread, the priority of the currently executing thread is kept for faster lookup in the global kernel information data structure. Both changes to the system state in the function body of `p4_runner_changeprio` have to be made without interference of other kernel threads – this is achieved by restricting interrupt handling during these state updates (via calls to `p4arch_disable_int` and `p4arch_enable_int`).

This call has a rather simple functionality, but it serves very well as an example because its execution spans all levels of the PikeOS microkernel, from high-level kernel functionality to hardware-related levels and the user-level interface (system calls are invoked via user interrupts). System calls with more complex functionality still span the same levels.

Abstract Kernel and Hardware Model

Because system calls are at the user's interface to the kernel and the PikeOS system is multi-platform, the kernel's specification has to hide any PowerPC implementation details to ensure proper encapsulation. The state of the PowerPC hardware on the machine level is reified with the help of a specification structure `PPC_c` containing, e.g., the contents of all relevant registers usually invisible on the C level – this corresponds to the abstract version of the `proc_t` C data structure introduced for the baby hypervisor

```
P4_uint32_t p4syscall_fast_set_prio(P4_uint32_t prio)
{
    P4k_thrinfo_t *proc;
    P4_uint32_t oldprio;

    proc = p4_runner();
    if (prio > proc->mcprio)
        prio = proc->mcprio;

    oldprio = p4_runner_changeprio(proc, prio);
    return oldprio;
}

P4_prio_t p4_runner_changeprio(P4k_thrinfo_t *proc,
                               P4_prio_t newprio)
{
    P4_prio_t oldprio;
    P4_cpureg_t oldstat;
    oldstat = p4arch_disable_int();
    oldprio = proc->userprio;
    proc->userprio = newprio;
    ... //update global kernel information
    p4arch_restore_int(oldstat);
    return oldprio;
}
```

Listing 3.2: Implementation of the PikeOS system call to change thread priority (`p4syscall_fast_set_prio`)

specification to model the VAMP architecture and is based on the register model as defined in the G2 PowerPC Core reference manual [Fre03]. For the externally visible components of the system we introduced an additional specification data structure, called `abstractModel` with an excerpt of the definition relevant for our system call example shown in Listing 3.3.

In our modeling, the intended encapsulation of the PowerPC state and the coupling invariants mandate that the abstraction of the kernel's state in ghost state, given by data structure `abstractModel`, owns the PowerPC machine model `PPC_c` as formalized by the ownership invariant `\mine(currentThread, &PPC_c)`.

For verifying the `p4syscall_fast_set_prio` system call, two components of the abstract model are needed: `interruptsEnabled`, which indicates whether the system currently allows external interrupts to occur, and a pointer to the thread currently running in kernel mode that is given by `currentThread`. These fields of the abstract model are related to the hardware and hence its representation as the ghost structure `PPC_c`.

```
1 _(ghost struct absModel_str {
      \bool interruptsEnabled;
      _(invariant interruptsEnabled == (PPC_c.msr.fld.EE == 1))
4     struct P4k_thrinfo_t *currentThread;
5     _(invariant \mine(currentThread, &PPC_c))
      _(invariant currentThread != NULL && ...)
  } abstractModel;)
```

Listing 3.3: Excerpt of abstract PikeOS model (C data structure definition)

Whether external interrupts are allowed or disallowed in the kernel is indicated by the field `PPC_c.msr.fld.EE` in the global ghost state model of the PowerPC hardware. In `abstractModel`, interrupts are defined to be enabled, iff this bit in the hardware model is set to 1, as stated by the invariant in line 3 of the specification of `absModel_str`. The value of pointer `currentThread` of C data structure `abstractModel` is related to the PowerPC hardware state by the register designated as stack pointer by the Application Binary Interface (not shown here for simplicity of the example).

The specification of the C methods on the upper layers of the kernel, like system calls, can now be written in terms of the elements of the abstract model.

System Call Verification

We now consider the annotations for the requirement specification, as well as the auxiliary annotations needed to verify the code of the system call under consideration. We start bottom-up with function `p4_runner_changeprio` that sets the data structures of the thread and the global info data structure of the kernel (called `kglobal`) and then move up to one of the callers of this method (the system call function `p4syscall_fast_set_prio`).

Listing 3.4 shows the requirement specification and (auxiliary) annotations necessary to verify functional correctness of the `p4_runner_changeprio` helper function.[2]

Before calling method `p4arch_disable_int` in line 12 to disable handling of external interrupts, which modifies the hardware model `PPC_c`, structure `abstractModel` has to be unwrapped – according to the ownership relation between both data structures, changes to `PPC_c` in this sequential setting are allowed only if structure `PPC_c` is mutable, which includes that this structure is owned directly by the thread.

After `p4arch_disable_int` has set the EE bit of the MSR variable in `PPC_c` to 1, the coupling invariant between `abstractModel` and `PPC_c` in line 3 of Listing 3.3 no longer holds. Before `abstractModel` can be wrapped again (line 23), this invariant has to be restored. This is achieved by a ghost statement that explicitly updates the `interruptsEnabled` flag of `abstractModel` (line 21). After the interrupts are disabled, the different updates

[2]The VCC annotation `_(maintains` *bexpr*`)` used in this listing is an abbreviation for the two annotations `_(requires` *bexpr*`)` `_(ensures` *bexpr*`)` where *bexpr* is a boolean VCC expression. The annotation `_(returns` *expr*`)` is equivalent to `_(ensures \result ==` *expr*`)` where *expr* is an expression of appropriate type.

```
 1  P4_prio_t p4_runner_changeprio(P4k_thrinfo_t *proc,
                                   P4_prio_t newprio)
        _(writes &abstractModel, &kglobal)
 4      _(requires proc == abstractModel.currentThread)
 5      _(maintains \wrapped(&abstractModel)
                 && \wrapped(&kglobal))
        _(ensures proc->schedprio == newprio && ...)
        _(returns \old(proc->userprio))
 9  {
10      P4_prio_t oldprio; P4_cpureg_t oldstat;
        _(unwrapping &abstractModel) {
12          oldstat = p4arch_disable_int();
13          _(ghost abstractModel.interruptsEnabled = 0;)
14          _(unwrapping proc) {
15              oldprio = proc->userprio;
                proc->userprio = newprio;
                ...;
            }
            ... //update global kernel information
20          p4arch_restore_int(oldstat);
21          _(ghost abstractModel.interruptsEnabled =
22                      PPC_c.msr.fld.EE;)
        }
24      return oldprio;
25  }
```

Listing 3.4: Annotations for verifying functional correctness of helper function to set the priority of a thread (`p4_runner_changeprio`).

on the priority values of the thread and kernel information data structure (left out for clarity in the code) can be performed (lines 14–19). Restoring the interrupt-enabled state (lines 20–21) and returning the old priority of the thread complete this method.

Using VCC it is now possible to prove that the function satisfies its specification given in lines 3–8. For this, about ten lines of intermediate assertions before/after calls to helper functions are necessary to let the verification system validate that certain properties have been preserved during method calls. In this case, these annotations are easy to find and assert properties of ownership relations between objects involved in the implementation (we have omitted these "lemmas" for brevity).

Following our bottom-up approach, the next function to consider is `p4syscall_fast_set_prio`. The implementation of this function first retrieves the kernel data structure for the thread with the help of the stack pointer part of the host CPU state using `p4_runner`. The specification of `p4_runner` abstracts from the concrete value of the stack pointer and the associated thread data structure and instead returns the ghost variable `abstractModel.currentThread`. This abstraction is valid as it is a system invariant (and part of *impl-inv* introduced in Section 3.1.2) that the stack pointer for the kernel stack always points to the memory page corresponding to the current thread. The current thread's data structure is placed at the beginning of this particular page.

This, finally, allows us to verify the following method contract for our exemplary system call `p4syscall_fast_set_prio`, which is identical to the contract of the helper function `p4_runner_changeprio` except for the behavior of limiting the range of input parameter `prio`:

```
1 P4_uint32_t p4syscall_fast_set_prio(P4_uint32_t prio)
      _(writes &abstractModel, &kglobal)
3     _(maintains \wrapped(&abstractModel)
               && \wrapped(&kglobal))
5     _(ensures prio <= abstractModel.currentThread->mcprio ?
6         abstractModel.currentThread->schedprio == prio && ...
        : abstractModel.currentThread->schedprio ==
             abstractModel.currentThread->mcprio && ... )
```

The postcondition of this method directly matches the informal specification in the kernel reference manual. Besides this postcondition, the contract specifies that the method is (only) allowed to write to `abstractModel` and `kglobal` (line 2), and that these two data structures are required to be wrapped according to the ownership methodology of VCC before and after the call to the function, i.e., the thread that is currently executing the method is in possession of these data structures, all their non-volatile fields remain unchanged and all their invariants hold.

Of course, as already mentioned, this system call specification tells only half the story – interference of concurrently executed kernel functionality has to be taken into account for a truthful representation of the system's behavior. The VCC verification methodology for the concurrent case is given in the next sections.

3.2.3. Concurrency

For the scalability of the verification task to large, *concurrent* software systems, modularization is vital. The VCC methodology supports this by being able to verify a method in a concurrent setting without imposing a fixed scheduling or having to inspect all possible traces through the system (similar to the modularity provided by, e.g., the rely-guarantee reasoning of Jones [Jon83]). The effects of the environment of the currently running thread are rather described by transitions on shared data structures. The modular verification technique of VCC for concurrent programs is based on a coarse scheduling of other threads, s.t. changes of other threads manifest themselves only just before the currently verified thread accesses shared state.

For proving simulation using VCC in the concurrent setting as described subsequently, we adopt the general verification methodology described by Cohen et al. [Coh+09b]:

> “Instead, we prove concurrent simulation . . . , by representing the abstract target with ghost state. The coupling invariant is expressed as an ordinary (single state) invariant linking the concrete and ghost state, and the specification of the simulated system is expressed with a two-state invariant on the ghost state. These invariants imply a (forward) simulation, with updates to the ghost state providing the needed existential witnesses [for the simulated state corresponding to the concrete state].”

In the following, we will show how we adapted this verification technique to the PikeOS setting with (restricted) kernel preemption in Verisoft XT and demonstrate the specification set-up needed for the verification of our running example of the PikeOS system call.

Modeling Concurrency in PikeOS

In our verification setup, concurrency originates only from interrupts and explicit scheduling by the kernel, as the system under verification has only a single processor. For performance reasons, not only processes running in user mode are preemptible, but also kernel threads may be interrupted at certain points of the execution. Access to the shared state by multiple user processes has to be rigidly controlled by the kernel in order to prevent race conditions.

To enforce a certain access policy to shared resources, the kernel is designed to use the following access mechanisms:

1. Reading and writing of data is done through locks with arbitrary locking granularity to achieve high performance by locking only the relevant state of the system that may be accessed concurrently.

2. Preemption of the kernel is prevented by either using hardware features to disable external interrupts of the system completely or by enforcing that the scheduler of the kernel is not invoked.

3. Memory separation, e.g., if it is known that all concurrently running processes can each only access a distinct part of the system, as common in interrupt handlers.

4. Using lock-free algorithms to access shared memory.

In the following, we discuss how to model these access mechanisms with VCC to reason about access to system state done through these mechanisms as in a sequential setting.

For the first access mechanism, namely locks implemented in the kernel, we re-use the verification methodology for locks provided in the work by Hillebrand and Leinenbach [HL09] that is available for VCC, in parts shown in Listing 3.5 and Listing 3.6. In the lock data structure definition given in Listing 3.5, the single object that is protected by the

```
typedef _(claimable) _(volatile_owns) struct _LOCK {
    volatile int locked;

    _(ghost \object protected_obj; )
    _(invariant locked == 0 ==> \mine(protected_obj))
} LOCK;
```

Listing 3.5: Lock data structure specification, modified excerpt from the work by Hillebrand and Leinenbach [HL09]

lock is stored in the specification field `protected_obj` – access to this object is controlled by the volatile field `locked` that is concurrently read and written by the different threads competing for the lock. The interface of the lock structure consists of the functions `Acquire` and `Release` with their VCC contracts shown in Listing 3.6.

Acquiring a lock using this specification grants the currently running thread exclusive access (on the specification level) to the object protected by the lock: The postcondition guarantees that the object protected by the lock is `wrapped` and `fresh`, i.e., it implicitly is contained in the writes set of the function calling `Acquire`. This makes it possible to reason sequentially about access to this object afterward. The corresponding implementation of `Acquire` is blocking, thus its contract only requires a claim for the lock object, but is not reliant upon the `locked` field of structure `LOCK` to be non-zero in the precondition.

Releasing the object protected by the lock requires the current thread to be owner of the object and also demands the object to be in a consistent state (using `\wrapped` as precondition). Also, to be able to add the protected object back to the lock, we need to know that the lock stays closed (hence the claim needed), otherwise, changes to the volatile field `locked` would not be possible. In addition, we need a non-aliasing property between this claim and the object to be protected by the lock. As a consequence of the possible concurrent execution of multiple threads trying to acquire the lock just after it has been released, the postcondition of `Release` cannot guarantee anything about either the protected object or the lock.

Similarly, these specification primitives are used to handle the second variant of access control to shared resources by limiting concurrency. A typical way to get exclusive access to memory on single processor systems is by restricting the sources of interrupting events. Recall that for PikeOS on a single-CPU system, there are two levels of access restriction by limiting concurrency: (a) disabling external interrupts altogether and (b) the alternative of just disabling preemption by the kernel (i.e., no other user process is scheduled by the kernel, but hardware interrupts are still handled).

Disabling external interrupts (e.g., the timer interrupt) on a single processor machine ensures sequential execution of the currently running kernel thread, by means of the hardware, until interrupts are enabled again. This behavior corresponds (on the specification level) to a locking scheme with a mandatory lock, i.e., the other threads of the system do not have to cooperate in order to enforce the lock. This makes it possible to

```
void Acquire(LOCK *l _(ghost \claim c))
    _(always c, l->\closed)
    _(ensures \wrapped(l->protected_obj) && \fresh(l->protected_obj))
{ ... }

void Release(LOCK *l _(ghost \claim c))
    _(always c, l->\closed)
    _(requires l->protected_obj != c)
    _(writes l->protected_obj)
    _(requires \wrapped(l->protected_obj))
{ ... }
```

Listing 3.6: Lock operation contracts, taken from the work by Hillebrand and Leinenbach [HL09]

model this access policy using VCC with the same locking mechanism used for ordinary locks, by transferring exclusive access of the whole state to the currently running thread for the duration that the interrupts are disabled. In the source code of the kernel, the primitives that disable and enable these interrupts are annotated with ghost code that performs the necessary transfer of exclusive access via ownership.

Specification of System Calls Revisited – Concurrent Execution

If a user process invokes kernel functionality via system calls by using the `sc` PowerPC assembly instruction, this causes the control flow to jump to the appropriate interrupt vector residing in a fixed location in physical memory address space. From this point on, the code runs in privileged mode with interrupt handling disabled. The kernel system call handler is responsible for saving relevant parts of the user context, enabling a return to user process execution after the system call has finished. After saving the user context, the system call handler invokes the appropriate C method in the kernel that implements the functionality of the system call.

Sequential execution of the assembly code of the interrupt handler up to the call of the C system call function allows giving a precise precondition relating parameters of the function to guest machine state at the point of system call invocation. After the system call handler has set up the state for execution of the actual system call, kernel execution is again interruptible. In this concurrent case, specifying code with pre- and postconditions is inappropriate: from a state that satisfies the precondition of a concurrently executed method, not only the effect of the system call has to be taken into account when writing the postcondition, but also an arbitrary number of steps of the environment.

Therefore, the specification of system calls is provided as definitions of transitions (two-state invariants) on the abstract model at the top-level of the system (part of cvm in Formula 3.1). The effect on the real hardware and implementation state of the kernel is then propagated via coupling invariants between abstract and concrete system state.

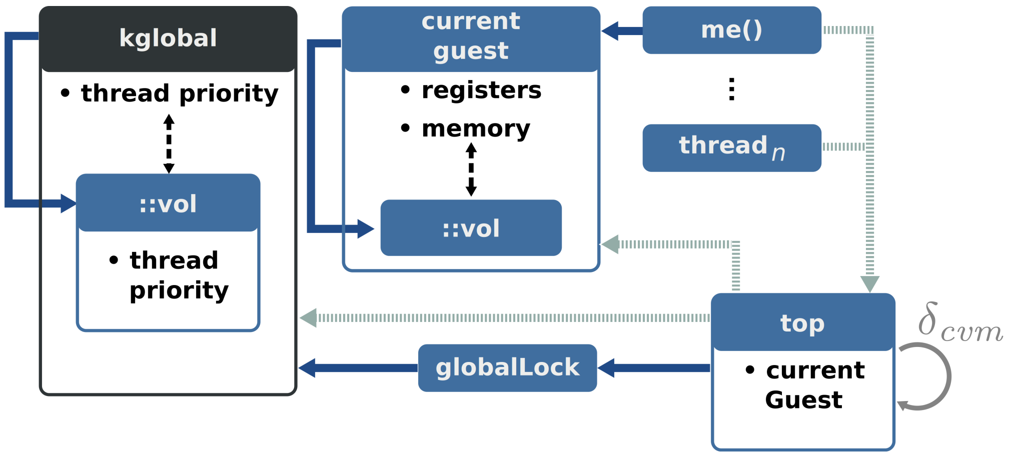


Figure 3.7.: Overall PikeOS specification structure. Black boxes denote C implementation state, blue boxes are used for specification data structures. Blue, solid arrows indicate ownership relations; green gray, dashed arrows signify claims held on the object pointed at; black, dashed arrows identify coupling invariants.

For our simple system call example, the setup of the specification structures and their relations to the actual kernel state is shown in Figure 3.7: At the top of the ownership hierarchy for the implementation data structures is `kglobal` (part of the kernel implementation state in Figure 3.1 on page 40), and as shown in excerpts in Listing 3.8.

```
struct {
  int currPrio;
  _(group _(claimable) vol)
  _(ghost _(:vol) volatile int volPrio;)                        }⟨VOL⟩
  _(invariant \mine(\this::vol) && currPrio == volPrio)

  _(invariant :vol (\approves(\this->\owner, volPrio))          }⟨ADM⟩
                  && (\same(volPrio) || \inv2(&top)))
} kglobal;

```

Listing 3.8: Annotated PikeOS global kernel information data strucure.

In this example, we omit the thread data structures our system call operates on and concentrate on changes to the global kernel information for simplicity. To be able to restrict updates to the kernel state according to two-state invariants, the (non-volatile) fields of structures below `kglobal` are duplicated using volatile ghost fields and coupled to their C counterparts with invariants (block labeled $\langle VOL\rangle$ in Listing 3.8). In this case the single field `volPrio`, representing the priority of the currently running thread is coupled to field `currPrio` in the implementation. For convenience, all ghost fields are collected in a VCC group (called `vol` in our example), which allows gathering different fields of a C structure under a common name similar to a new sub-structure definition. Access to `kglobal` has to be coordinated among the different kernel threads and is thus put under the control of a lock (`globalLock`) which holds ownership of `kglobal`.

```
_(ghost _(claimable) struct {
  pGuest curG;
  _(ghost \claim ct, cg;)
  _(invariant \claims(ct, (&kglobal)::vol->\closed)
           && \claims(cg, curG::vol->\closed))
  _(invariant \mine(ct, cg, &globalLock)
           && globalLock.protected_obj == &kglobal
           && &top == \this)

  _(invariant (\same(kglobal.volPrio) && \same(curG->volr1)
            && \same(curG->volr2) && \same(curG->volpc))
        || (curG->mem[\old(curG->volpc)] == SC &&
            kglobal.volPrio == \old(curG->volr1) &&              ⟨TR⟩
            curG->pc == \old(curG->volpc) + 1 &&
            curG->volr2 == \old(kglobal.volPrio)))
} top;)
```

Listing 3.9: Annotated top-level specification structure corresponding to cvm in Formula 3.1 on page 40.

Specification of kernel functionality is not given directly via invariants in the `kglobal` data structure, but rather collected in one place as two-state invariants of ghost structure `top`, as shown in Listing 3.9. The two-state invariant of `top` refers to volatile state of `kglobal`. For this construction to be admissible (cf. Section 2.4.1), structure `kglobal` includes the invariants in the block labeled $\langle ADM \rangle$ – the approver keyword [Coh+09a] and the second two-state invariant in this block have a similar effect on which change to `kglobal` is allowed: Either the volatile field storing the priority remains unchanged (`\same(volPrio)`) or the prover has to show that the top-level data structure is changed according to the transitions permitted by its two-state invariant (`\inv2(&top)`). For the same purpose, data structure `top` holds a claim on `kglobal::vol` which guarantees that only changes according to the group's two-state invariant are made by other threads.

All threads in the system in turn are given the guarantee that the other threads behave according to the two-state invariants given in `top`. This is accomplished by passing appropriate claims to the threads, the resulting state is illustrated in Figure 3.7: In the single-threaded initialization phase of the kernel, this claim is established (and never relinquished) and duplicated for each thread spawned later on. As there exists (at least) the one valid claim created at boot time, no single thread is able to take exclusive control over structure `top`, thus enforcing that this thread has to obey the two-state invariants.

In our example, the single two-state invariant of structure `top`, given in block $\langle TR \rangle$ of Listing 3.9, defines the effect of calling the system call `p4syscall_fast_set_prio` with the help of the global kernel information data structure, as well as a model of the simulated guest system (field `curG`). Again, as with `kglobal`, data structure `top` has to keep a claim to guarantee only changes according to its two-state invariant are allowed.

The ghost model for the guest systems uses the same specification pattern as the `kglobal` data structure by introducing a volatile copy of its state, coupled via invariants (see also Fig. 3.7):

```
typedef _(claimable) struct guest_str {
    int pc, r1, r2, *mem;
    ... //volatile copy with coupling invariants
} guest, *pGuest;
```

Two-state invariant $\langle TR \rangle$ of `top` postulates that either the state of `kglobal` and the current guest are unchanged (corresponding to a stuttering step), or the current guest invokes a system call (with the only implemented system call `p4syscall_fast_set_prio`), which in our simplified setting changes the priority of the currently executing thread and updates the guest state (updating the program counter and one register of the guest that stores the return value of our system call).

The initial and final states of such transitions are states of the system that are visible to the user processes, i.e., internal computations of the kernel are visible to the environment as one atomic state update, as long as no interrupt inside the kernel thread is possible. For the verification of system calls, with a structure as defined in the beginning of this section, there may exist several sequentially executed code blocks with interruptible code segments in between. In the interruptible code parts, the kernel may schedule other user threads running in user mode, which can witness the change of state due to the partly executed system call.

On the abstract level, such a system call would therefore correspond to several transitions, one for each non-interruptible code block in the system call. On a higher level, it has to be shown that the effect of performing all of these transitions that constitute one system call, interrupted by transitions of the environment, has the desired functionality as described in the kernel API documentation.

Verifying `p4_runner_changeprio` Using the specification structure shown in Figure 3.7, as well as the lock mechanism to specify enabling and disabling interrupt handling, verifying our sample system call in the concurrent setting becomes straightforward. Listing 3.10 shows a function that emulates a simple version of invoking the system call via instruction issued by a virtual guest machine.

Taking the interrupt triggered by system call invocation proceeds in two stages: in the first phase, further interrupts are disabled, the current state is saved on the stack as far as needed to allow interrupts to occur afterward and control is passed to the right system call handler – function `syscallHandler` in Listing 3.10. The portion of the stack storing the guest state (as described by the $\mathcal{B}$-relation in Section 3.1.2) is only accessed by the thread responsible for handling the system call – as a consequence, we assume the associated interpretation of the stack values as `guest` data structure given as parameter to function `syscallHandler` to be `wrapped` without any outstanding claims on it. We also require in the precondition of the function that this guest data structure is identical to the pointer to the current guest of the top-level abstract state. Claim `c`, passed to the function, reifies the guarantees about `top` given to every thread of the system.

As the effect of the concurrent execution of the system call is captured by the two-state invariant of `top`, the method contract of function `syscallHandler` neither provides any information about the value of the system call parameter stored in register `r1` of the guest nor the return value.

In the function body of `syscallHandler`, interrupts are enabled again, so before accessing the global kernel state, interrupt handling has to be disabled, modeled by acquiring `globalLock`, which gives exclusive access to data structure `kglobal`. Afterward, the priority of the current thread can be changed in the kernel information structure, and the guest is updated according to the semantics of the system call instruction. That these updates conform to the two-state invariant of `top` is shown by performing a valid transition of the volatile system state in ghost updates in block $\langle UP \rangle$. Releasing the global lock that models disabling interrupts of the system completes the implementation of the function.

The PikeOS Simulation Proof

To demonstrate functional correctness of the PikeOS kernel according to the simulation proof technique as introduced in Section 3.1.2, we have to show that – starting from a pair of initial concrete system state and abstract state fulfilling predicate *cvm-sim* – after some steps of the hardware, a concrete state is reached that again has an abstract counterpart w.r.t. *cvm-sim*.

Similar to the simulation pattern for the sequential setting of the baby hypervisor verification by Alkassar et al. [Alk+10c] introduced in Section 3.2.1, we can emulate execution of the hardware by calling a step function processing the current instruction (or interrupt) of the modeled system in a while loop.

With the specification approach for concurrent kernel execution as presented in the previous section, two-state invariants on the abstract state guarantee that only valid transitions are performed by the kernel implementation. In the user simulator for concurrent system simulation, the modeled hardware state `h` is coupled via invariants to the abstract state stored below structure `top` – thus to show our simulation property, a claim that `top` stays closed guarantees correct operation of the overall system:

```
sim_step(h)
while(1)
    _(invariant I && \always(c, &top->\closed))
{
    sim_step(h)
}
```

```
void syscallHandler(pGuest g _(ghost \claim c))
  _(requires \wrapped0(g)
          && top.curG == g && ...)
  _(always c, (&top)->\closed)
  _(writes g)
{
    if (g->mem[g->pc] == SC) {
        int oldSharedVar = kglobal.currPrio;

        Acquire(&globalLock _(ghost c));
        _(unwrap &kglobal)
        kglobal.currPrio = g->r1;
        _(unwrap g)
            g->pc = g->pc + 1;
            g->r2 = oldSharedVar;

            _(atomic &kglobal::vol, c, g::vol) {                    ⎫
              _(ghost kglobal.volPrio = kglobal.currPrio;)          ⎪
              _(ghost g->volr2 = g->r2;)                            ⎬ ⟨UP⟩
              _(ghost g->volpc = g->pc;)                            ⎪
            }                                                       ⎭
        _(wrap g)
        _(wrap &kglobal)
        Release(&globalLock _(ghost c));
    }
}
```

Listing 3.10: Annotated implementation for verifying `p4_runner_changeprio` in the concurrent setting.

3.3. Ingredients for Pervasive Correctness

Correctness of the verification results of PikeOS on the C layer, as shown in the previous sections, relies on various properties not only of the underlying system (hardware as well as software), but also on the tools and methodology used for verification. In the following, we highlight some building blocks on which our verification technique on the abstract C level rests – for a more detailed account in the PikeOS setting, we refer to our previous work [Bau+10]. Similarly, Cohen et al. [CPS13] provide a comprehensive report on the different ingredients of a theory of multi core hypervisor correctness in the context of the verification of Hyper-V. Most of the components in this theory are also relevant in the PikeOS verification, although our system setup is simpler in some regards due to the single core hardware and the paravirtualization setup (as a consequence, e.g., store buffers are transparent and there are no memory races on the translation lookaside buffer).

Besides considering only a small fraction of the functional correctness proof for a virtualization platform like PikeOS in this thesis, non-functional requirements like security or efficiency are out of the scope of this work. For the verification of PikeOS, we also relied on the correctness of most components involved – such as the C compiler, the VCC verification tool or the underlying PowerPC hardware – to be able to focus on the verification of the PikeOS implementation.

For a pervasive correctness proof, one prerequisite would be formal computation models on the different abstraction layers, starting with the hardware at gate level, to the semantics of the instruction set architecture, up to the formal semantics of C and assembler code – together with simulation proofs relating these different abstraction layers, as outlined by Cohen et al. [CPS13].

Using these sequential simulation proofs between the different layers, Baumann [Bau14] demonstrates how to prove a simulation theorem for the pervasive system verification in the *concurrent* setting. For this, Baumann introduces a model of "concurrent system with shared memory and ownership" (COSMOS) that can be used to describe (concurrent) systems at different abstraction levels. With the help of a reordering theorem (which allows treating blocks of computations of a single machine instead of arbitrary interleavings of all machines in the model), consistency relations between the different abstraction layers from the sequential simulation proof are applicable to the sequential blocks of the execution. The ownership policy as part of the COSMOS model ensures absence of memory races and allows transferring safety properties from the abstract to the concrete level.

Another essential ingredient for pervasive system correctness is the soundness of the VCC tool, as well as that the underlying, implicit semantics of C (plus ghost state and code) is compatible with the C semantics of the compiler used to translate the C implementation of the verified kernel to machine code. Soundness of VCC depends on the correct translation from annotated C code to the intermediate Boogie language (and thus in turn on the soundness of the Boogie tool), but also on the background axiomatization used by VCC. To improve trust in the latter, we introduce an approach to test the axiomatization in the second part of the thesis in Chapter 6.

For the verification result of VCC to be meaningful, the verified implementation and underlying system also have to fulfill certain assumptions (e.g., no self modifying code or a sequentially consistent memory) – that these assumptions are met may then lead to additional proof obligations on the concrete implementation, e.g., as shown for the store buffer reduction theorem by Cohen and Schirmer [CS10].

One issue we did not cover so far is the interaction between the different language levels involved in executing concurrent system software, i.e., in our case between C code and (inline/macro) assembly. For the verification of implementations consisting of both C and assembly functions for PikeOS, the approach used was to translate assembly instructions to C functions operating on a ghost model of the PowerPC hardware, which allowed proving the resulting C implementation using the VCC tool [Bau+09b].

Another example for interaction between layers of the semantic stack is the thread switch in the kernel, where C execution is disrupted by changing the execution context via assembly instructions: a context switch in operating systems can be implemented through the two methods setjmp and longjmp – former stores the hardware context of the running thread in kernel memory, while latter restores a previously saved process from kernel memory to hardware. A thread that is scheduled through longjmp restarts its execution right after the setjmp method where it formerly saved its CPU context.

For efficiency, reasoning about C programs with VCC is, however, performed on an abstract level that does not model the concrete effect of executing the C instructions on the real hardware. Due to the modularity of verification, even if there was a coupling relation between C- and hardware-level within VCC, the view is restricted to the current thread and the method under verification. The contexts of previous and concurrent method invocations are invisible to the verification engineer. Therefore, we have to model the context switch implicitly using VCC's built-in features.

The actual (assembly) implementations of the process save and restore mechanisms as described above each have a corresponding transition on the system state and can be verified using the assembly verification technique described before. In the currently running kernel thread that performs the context switch we have, however, a gap between loading the context of the next thread to be scheduled and eventually returning to the original thread via another context switch. Between those two points, an arbitrary number of steps of the other threads of the system are performed. This can be modeled on the specification level by giving up exclusive access to the system state at the point when switching the contexts. From the caller site, the method that switches the contexts thus performs an arbitrary number of transitions on this state, according to the system transitions defined by two-state invariants.

One of these transitions describes the effect of restoring one of the stored CPU contexts of the threads in the system. If we assume a fair scheduler, eventually one such transition will be performed that restores the initial thread that performed its process save. This justifies handling the context switch as a (special kind of) method invocation from which control flow eventually returns.

3.4. Related Work

Verification of system software has a long tradition. Providing a full account of the different efforts to formally prove correctness properties for (parts of) operating systems warrants its own treatise, for which we refer to the comprehensive overview by Klein [Kle09] – this paper spans early verification attempts which go back to the 1970s and 1980s (with UCLA Secure Unix [WKP80], PSOS [FN79] and KIT [Bev89] as the projects mentioned), up to the recent Verisoft and L4.verified projects. In addition, the related work section of the paper by Klein et al. [Kle+14] includes more recent developments in this research area.

Here, we take a look at two prominent large-scale verification efforts that deal with system software correctness proofs on source code level: the L4.verified project [Kle+10] and the Hypervisor project [LS09] within Verisoft XT. In the following, we will give an overview of the projects, their verification targets and goals and a brief summary of the verification methodologies used.

3.4.1. The Hypervisor Verification Project of Verisoft XT

The verification target of the Hypervisor sub-project within Verisoft XT was the kernel of Microsoft's virtualization solution Hyper-V (in the following, we use the name Hyper-V and Hypervisor synonymously for the kernel of Hyper-V). During the Verisoft XT project from 2007–2010, the European Microsoft Innovation Center, Microsoft Research, the German Research Center for Artificial Intelligence, and Saarland University worked on the specification and verification of the Hypervisor, as well as the development of the verification tools and methodologies needed to accomplish the Hypervisor correctness proof [LS09].

Hyper-V is a multi-core capable hypervisor for x64 – as a native hypervisor, it runs directly on the x64 hardware offering full virtualization by providing virtual x64 machines augmented by *hypercalls* to access virtual machine management functionality [LS09]. Compared to PikeOS, the Hyper-V source code is substantially larger with about $100\,\mathrm{k}$ lines of C code and about $5\,\mathrm{k}$ lines of assembly [LS09]. Also, Hyper-V makes use of a multi-core hardware setup; providing full virtualization instead of paravirtualization complicates memory translation, as the hypervisor has to emulate the host MMU. Similar to the setup of PikeOS, the code base of the hypervisor was not adapted to simplify verification. Concerning the amount of annotations per line of source code needed, a maximum ratio of three to one is reported in presentations.

Correctness of the hypervisor implementation is shown with the help of a simulation proof using VCC – a short summary of the VCC verification technique for the Hypervisor can be found in the work by Leinenbach and Santen [LS09]. The general VCC specification methodology of the Hypervisor has been adopted by us for the verification of PikeOS, as described in Section 3.2 for the concurrent case.

Concerning the verification progress of the hypervisor, Leinenbach and Santen [LS09] state: "As of July 2009, the Hypervisor verification is still ongoing. Data structure invariants are in place and the larger part of public interfaces is specified. Several hundred functions have been verified with VCC." Further, Cohen et al. [CPS13] describe the state of the verification effort after the project duration: "When the project ended in 2010, crucial and tricky portions of the hypervisor product were formally verified [...]". In the context of Verisoft XT, using auto-active verification with VCC, research results include the verification of a shadow page table algorithm [Alk+10a], verification of interprocess communication [Alk+10b], and verification of TLB virtualization code [Alk+12].

Besides verification of the Hyper-V kernel, the functional correctness proof of a simple version of a hypervisor – the baby hypervisor, as described in Section 3.2.1 – has been accomplished [Alk+10c; PSS12] showing feasibility of auto-active verification for system software. Specification effort mentioned in [Alk+10d] matches the ratio reported for Hyper-V: the implementation consisted about $2.5\,\mathrm{k}$ C code tokens compared to the $7.7\,\mathrm{k}$ annotation tokens needed. Based on experience gathered within verification of those hypervisors, the VCC methodology and tool chain could be improved [Alk+10d] with the goal to increase performance of the automated prover and especially to guarantee fast response times in case of failed verification attempts, as this is the common case in software verification.

3.4.2. The L4.verified Project

In the L4.verified project [NIC11b], source level verification of the seL4 microkernel [NG14] was conducted, resulting in correctness proof of the kernel in 2009 [Kle+14].

In several research projects with a combined duration of "more than 8 years in total" [Kle+14], besides this functional correctness proof of the operating system kernel, which was achieved by proving refinement between the concrete C code and an abstract specification [Kle+10], also further high-level properties on top of this abstract specification could be established (one example is non-interference [Mur+12]). In addition, a refinement proof between the C layer and the machine code executed on the hardware allowed to transfer the correctness properties through the whole system stack [SMK13].

Compared to the goals of the first Verisoft project, the seL4 correctness proof is not completely pervasive in this sense, as it assumes correctness "of TLB and cache flushing operations as well as the correctness of machine interface functions implemented in handwritten assembly" [Kle14], besides correct operation of the hardware.

The L4-based microkernel with a size in the order of 10 kLOC was developed from scratch with the formal correctness verification task in mind – as reported by Klein [Kle09] for the development of seL4, feedback from the L4.verified project influenced the design of the kernel originating from the seL4 developers. However, Klein [Kle14] points out that "it was a strict requirement of the project not to sacrifice critical runtime performance [of seL4] for ease of verification".

Since 2014, the seL4 microkernel is distributed under open source licensing terms and is available at `http://sel4.systems/`, together with the correctness proofs.

In the L4.verified project, the approach taken to verify seL4 differs in various ways from the one used for the PikeOS and Hyper-V verification in Verisoft XT. Besides using the interactive theorem prover Isabelle/HOL for specification and interactive verification of the seL4 microkernel, the verification target was written from scratch and could be customized to a certain extent to simplify the proof task. One prominent example for this is to control concurrency in kernel execution – Klein et al. [Kle+14] note that "By design, we side-step addressing the verification complexity of yield by using an event-based kernel execution model, with a single kernel stack, and a mostly atomic application programming interface [...]."

Also, an abstraction layer in form of a Haskell implementation of the kernel has been reported to be of great help (instead of directly verifying the equivalent C implementation). From this Haskell version of the kernel, an executable Isabelle/HOL specification is automatically generated that is verified to correspond to a (manually written) abstract specification via a refinement proof. Another refinement proof then establishes correspondence between the executable specification and the C implementation. Both implementations in Haskell and C together amount to approximately 15 kLOC, while the Isabelle script used in the refinement proofs for functional correctness spans 165 kLOC [Kle+10]. Klein et al. [Kle+14] provide an overview of the different phases in development of the seL4 implementation and verification and state that the effort for the seL4 refinement proofs amounted to 11 person-years.

4. Lessons Learned From Microkernel Verification

In the previous chapter we have shown how verification of a microkernel correctness property for a real-world preemptible kernel can be accomplished with the help of an auto-active verification system like VCC. While the specification for the simplified examples given at an abstract level could be kept concise, for the real verification target, however, often complex contracts and auxiliary annotations are necessary to complete verification, e.g., due to implementation details. Even when the resulting annotations needed for successful verification are all worked out and lead to an elegant specification, this does not represent the often time-consuming process involved to reach the final set of annotations.

We will take account of the process of coming up with a practicable specification in this chapter and discuss the challenges and issues encountered within the specification and verification of a large-scale software project. Much of the work of system software verification as presented in Chapter 3 rests on results achieved previously, e.g., the model of communicating virtual machines developed in the predecessor project Verisoft, or the specification and verification methodologies conceived by the VCC developers, as well as the verification team of the Verisoft XT Hyper-V project. One of the contributions of this thesis is thus not so much as to establish a new specification methodology for use in microkernel correctness but rather to pinpoint and (partly) address issues encountered in specification and verification of an industrial software system – the resulting insights have been enabled by the PikeOS case study. Most of the issues presented further are addressed by first ideas how to improve the tools or processes involved. For some of these challenges, we give a more detailed treatment in Part II of this work.

Although the list of issues presented in this section might indicate otherwise, our primary conclusion from the Verisoft XT project is that current auto-active verification systems are powerful enough to be successfully applied to concurrent microkernels. For VCC-based verification in Verisoft XT we focused on core parts of the microkernel, and all relevant specification mechanisms could be established.

Independently of the size and type of the system to be verified, the verification task using annotation-based verification tools can be divided into the following three phases:

1. Formalization of given (informal) requirement specifications as program annotations.
2. Adding auxiliary annotations to describe the boundaries and interfaces of the different modules of the system (e.g., function contracts or loop invariants).
3. "Local" verification of single modules (functions) in isolation.

In practice, all three steps have to be performed repeatedly during several iterations of changing annotations until a fixpoint is reached, any bugs in the code or the requirement specification are corrected, and verification of the whole system succeeds.

There are several common ways to simplify the verification of large and complex systems and make it feasible in practice: (a) reducing the cost of specifying and verifying a single property of a single module (function), (b) modularization, i.e., decomposing the verification task by verifying one module of the system at a time, and (c) abstracting from details of the system's implementation and behavior.

All three of these concepts are addressed to a certain extent by current deductive verification tools: (a) Verification tools have made a leap forward in recent years, enabling users to verify individual functions once considered challenging with ease. (b) Annotation-based verification tools like VCC make use of decomposition of the verification task by verifying individual functions and threads modularly. Unfortunately, in practice, the verification effort does not scale linearly with the number of modules due to interactions via shared data structures and common parts of the program states. (c) Abstraction is possible using a separate specification state and abstract data types. However, support for this is limited in the VCC tool and methodology.

In the following, we will illustrate why verification of a system like PikeOS still is a challenging task, despite all support by the verification tool and methodology. Although some discussed issues are more prominent for verification targets that have the characteristics of a microkernel, they are in no way exclusive for such programs but occur to a certain extent with every large software system. The remainder of this chapter is structured according to the three specification and verification phases listed above.

4.1. Formalizing Requirements

As we will discuss in depth in Chapter 7, in theory it is not necessary to come up with top-level contracts adapted to the (often informal) requirement specification of the system, as the strongest possible contract must always suffice. However, in practice, the strongest contract not only makes verification more complicated, but it also obscures the intention behind the behavior of the system. In addition, regardless of the strength of a contract, the user has to find the right kind of abstraction when formalizing informal requirements. For these reasons, formal top-level contracts must take the informal requirements and other system documentation into consideration.

Issue A-1: Implicit Behavior in Informal Specifications. There is rich official material in form of end-user documentation and requirement engineering documentation for PikeOS. These are however focused on the strict separation of platforms, architecture, and kernel (which is justified from a maintenance perspective) and is thus of limited use for finding annotations for functional verification.

Example. An example for user-level documentation keeping concurrency effects implicit is the informal specification of the system call that changes the priority of a thread, as introduced in Section 3.2.2. We have seen that this system call is preemptible, and if during a preemption another user thread has changed the thread's priority before the function's return value is assigned, then the "old priority" returned might not be what a naive observer might expect who neglects that the system is concurrent.

Approaches to Resolving the Issue. Ideally, to simplify the transition from the more informal existing system documentation, a first step would be a specification mechanism that allows to write down the intention of the programmer and system architect explicitly, although without the need of a complex formal specification. Preferably, these documents have to be of value in the software development process besides formal verification. An example of such precise specification of the expected system behavior are test cases.

In this regard, also documents supporting certification measures are helpful: the DO-178 avionics certification requires, e.g., descriptions of concurrency and reentrance analyses. Also, the Common Criteria for Information Technology Security give suggestions to system architects in how to structure an architecture of a system and how to describe the security properties of a system in terms of domain separation, self-protection and non-bypassibility.

Besides existing documentation, code inspection is a practicable way to get a first version of the auxiliary annotations needed for the verification of the requirement specification. However, this involves the risk of repeating mistakes in the specification that were already introduced in the implementation.

Issue A-2: No Syntactic Distinction Between Different Kinds of Annotations. Besides the requirement specification, there are two kinds of auxiliary annotations needed for verification with VCC (essential and non-essential), as already briefly mentioned in Section 2.4.1. However, in the case of VCC, there is no clearly visible distinction between auxiliary and requirement specification and between essential and non-essential annotations added for performance reasons. We argue in Chapter 5 that it is extremely important for the user to have knowledge about which kind of annotations are essential for the verification system as without that knowledge they may continue to add the wrong annotations in case of a failed proof attempt.

Moreover, we claim that requirement and auxiliary annotations must be syntactically distinguished. That makes specifications clearer and easier to read and understand. In certification processes it is indispensable to have a very clear understanding of which annotations form the requirement specification that has been verified. Also, managing annotations in case of software evolution is more difficult when no clear distinction is given: while auxiliary annotations may be changed or removed at will, requirements have to stay unchanged. As a consequence, without this distinction, maintainability of the annotations suffers, e.g., by having to separate requirements from auxiliary annotations in such a case.

Example. For many software systems, large parts of the implementation and thus their contracts do not directly belong to the external interface respectively requirement specification of the system: examples include contracts of inner methods, as well as loop invariants. However, VCC relies on user-provided contracts for these modules due to performance reasons. For functions, this is also independent of whether the method is supposed to be inlined in the resulting compiled program.

An example from the PikeOS kernel is the implementation[1] to restore the condition of whether interrupts are admitted or not to an earlier saved state of the system, given by parameter `msr`:

```
static inline void p4arch_restore_int(P4_cpureg_t msr)
{
  unsigned ret, val;
  __asm__ ("mfmsr %0" : "=r"(ret));
  val = ret | (msr & MSR_EE);
  __asm__ ("mtmsr %0" : : "r"(val) : "memory");
}
```

This simple implementation uses two inline assembly statements to read and write a bit of the machine state register (MSR). The value to be written to this register is determined by a bitwise OR operation on the old value and a certain bit from parameter `msr`.

The contract of this function as given in the following simply replicates this state update – albeit on the specification structure `PPC_c` (as introduced in Section 3.2.2), representing the CPU state of the machine otherwise not explicitly visible on the C level:

```
static inline void p4arch_restore_int(P4_cpureg_t msr)
  _(writes &PPC_c)
  _(maintains \wrapped(&PPC_c))
  _(ensures PPC_c.msr.fld.EE ==
                \old(PPC_c.msr.fld.EE) | GET_EE(msr)
       && \unchanged(PPC_c.msr.fld.PR))
```

This contract not only duplicates information given concisely by the implementation and produces additional overhead in terms of framing and ownership annotations, but also lacks a clear labeling that these annotations are *auxiliary* and do not belong to the requirement specification, as this function is never called directly by a user process.

Approaches to Resolving the Issue. It is preferable to keep requirement and auxiliary annotations separate, e.g., requirement annotations in the header file and auxiliary annotations in the C source file. Where no such separation is possible, keywords (in the style of visibility modifiers `public` and `private`) should be used.

4.2. Adding Auxiliary Annotations

After the requirements have been formalized as annotations, a multitude of essential auxiliary annotations have to be added before even the first verification attempt can begin. Among these auxiliary annotations are function contracts, loop invariants, as well as data structure invariants. Usually, this initial set of auxiliary annotations is not sufficient to prove the part of the system considered correct and the annotations have to be adapted in several iterations, for reasons further explained below.

[1]Slightly changed implementation taken from the work by Baumann et al. [Bau+09b], adapted to current VCC syntax.

4.2.1. Modularization

Modularization helps to reduce verification effort by decomposing the verification task. For the modular verification of sequential programs with VCC, a function is verified using the contracts of all the functions it calls – instead of their implementations. This design choice relieves the load on the automated prover at the expense of having to write (auxiliary) function contracts for all functions.

For this reason, using current auto-active verification tools, the bottleneck of verification is how to find the right auxiliary specifications and specifications of interfaces between modules. In addition, architectures such as microkernels feature some inherent characteristics that restrict the extent to which the specification task can be modularized.

In the case of the PikeOS microkernel, implementations of single C functions are deliberately kept simple to facilitate maintainability and certification measures – the functionality of the whole kernel is rather implemented by interaction of many of these small functions, operating on common data structures. Microkernels, and more generally, all operating systems, have to keep track of the overall system's state, resulting in relatively large and complex data structures on which many of the kernels functions operate conjointly. This amount of interdependencies has an impact on the following issues.

Issue B-1: Entangled Specifications. Function specifications have strong dependencies on each other. Finding the right annotations for a single function requires the verification engineer to consider several functions at once, due to these dependencies. Good feedback of the verification tool in case of a failed verification attempt is essential. Feedback given by annotation-based verification tools so far only focuses on the function currently being verified. This allows the user to pinpoint and fix bugs in the specification or program respectively to change auxiliary annotations, e.g., loop invariants or contracts for called functions. For the verification of single functions, this tool support is sufficient. However, current verification methodologies often do not provide adequate assistance for the user in case of analyzing problems with interdependent specifications.

Example. One example for the interplay between specifications of different implementation parts due to dependencies between functions is examined in detail in Chapter 7.

Regarding the complexity of system software, Klein et al. [Kle+14] note that "...OS kernels ...generally feature highly interdependent subsystems" (citing Bowman et al. [BHB99]). That this is not an exclusive property of system software or microkernels in particular but usual in non-trivial software projects in general is demonstrated by empirical studies on software complexity metrics – for example, Bhattacharya et al. [Bha+12] examine some metrics of eleven open source software projects of significant size. One of the characteristics analyzed is the diameter of the static call graph (defined as "the longest shortest path between any two vertices in the graph"). For the projects examined, the authors note that for the diameter "the typical value range (10–20) is similar across all programs."

For concrete instances of function dependencies in PikeOS, we refer to the examples given in previous work describing our progress in kernel verification [Bau+11; Bau+09b].

Besides dependencies between different modules, also annotations within a single module occasionally have fragile interdependencies, e.g., nested or consecutive loops in a function that necessitate several adaptation steps of the different loop invariants. While the need for changes is often plausible when strengthening or weakening contracts to simplify verification of other parts of the program, the need for changes in the formulation or structure (often resulting in a semantically equivalent specification) might come as a surprise. One example is the need to provide appropriate triggers for the underlying SMT solver – this issue is complicated enough to warrant its own guide on finding the right triggers [Mos09].

Approaches to Resolving the Issue. One solution to this issue, we describe in detail in Chapter 7, is to provide early feedback when starting to specify a software system. Our approach is based on the combination of deductive verification as in VCC with software bounded model checking (SBMC) using the LLBMC tool [SFM10]. For this, annotations written in VCC's specification language are translated into assertions that can be checked by LLBMC (i.e., boolean C expressions extended with some features specific to LLBMC). The SBMC procedure then allows to check these assertions without the need to provide any additional auxiliary annotations as in case with VCC, e.g., functions are inlined and loops unrolled instead of modularized using annotations. However, in contrast to deductive verification, the number of loop iterations and the function invocation depth is bounded. If no assertion is violated within the given bounds, this does not imply that the program adheres to its specification, as a bug may still occur, e.g., in a loop iteration outside the given bound. That is, LLBMC does not provide a full proof (that is left to VCC) but a quick check that can point the user to problems in the annotations early on and thus avoid unnecessary VCC proof attempts.

Issue B-2: Module Granularity. Dependencies between functions obfuscate module boundaries and thus make finding the right module interfaces difficult. In addition, even if larger modules can be identified, there is no particular support for a hierarchical modularization in annotation-based verification systems and for specifying properties of such a larger module, e.g., on the system architecture level.

Also, by the VCC methodology, module boundaries are fixed by having to provide function contracts for all functions – this implies that every function is also a module. However, some function boundaries of the software system might also be chosen for reasons other than clear separation of disjoint functionality of different modules. Ideally, for those functions of a large software system that are not part of the externally visible part of the system's interface, such function contracts could be omitted – especially for small library functions that occur at the leaves of the system's call graph.

Example. Already the C language only provides a rather coarse concept of modules compared to, e.g., classes and packages in object-oriented programming languages like Java: visibility of functions in C by default spans the whole program (as long as the header files containing the function declaration are included) – the visibility scope can be restricted to the translation unit (approximately corresponding to a C file with all its includes) the function is declared in by using the storage-class specifier `static` [ISO+11].

Hence, without any further (specification) language features there is only limited means to convey the modularization concept the developer had in mind when implementing a set of functions other than the distribution among different C source files.

While the VCC specification language allows for more fine-grained modules with the definition of block contracts inside a function body, in the other direction, VCC treats each function as a module and there is no provision for further aggregation. In contrast, modules in PikeOS are not restricted to a single function but often coincide with the partitioning of functions into the different source code files. In addition, modules in concurrent systems may not coincide with functions but encompass statements executed in sequence that result in a single observable ("atomic") step of the system.

Approaches to Resolving the Issue. Inlining of function calls would help to make function contracts obsolete for those functions of the system that are not a module on its own but only part of some larger functionality. This would already moderate the issue of imposing fixed module boundaries for functions in some cases.

4.2.2. Abstraction

Besides modularization, abstraction is another important instrument to handle verification of large software systems. Good abstraction of the functional behavior of a system helps to focus on important details of the functionality, and allows for clear and succinct specifications. Poorly chosen abstractions may complicate verification up to making it impossible to verify the system at all. Which abstraction is appropriate not only depends on the system properties to be specified but also on how well the verification tool is able to reason about it.

Issue B-3: Finding the Right Abstraction. To find the right abstraction for data structures, analyzing the source code alone is often not sufficient in practice. Complexity of the implementation of a single data structure is secondary when considering verification effort. Gathering the important properties of the data structure is crucial to find the right abstraction. Which of these properties are needed and relied upon in the software depends on its usage in the functions of the system. While some techniques such as CEGAR exist that may help in some cases in finding the right abstractions, these functions are not sufficiently supported in current deductive annotation-based systems.

Again, multiple dependencies between the functions that all operate on (parts of) the same data structure make it hard to find the right abstraction. Besides better support from verification tools, to come up with the right abstractions, information from system developers and architects is vital.

Example. A simple example of the difficulties of finding the right abstraction are implementations of abstract data types. While certain data types suggest a canonical representation in the implementation (e.g., stacks implemented as linked list data structures), this correspondence is not fixed, moreover, also in the opposite direction the relation between the two levels of abstraction may be ambiguous.

For instance, a linked list data structure in the implementation may either correspond to a sequence as given by the pointers in the structure, or, when the order of elements is in fact irrelevant and the data structure is just used as a convenient way to store arbitrary many elements, encode just a set of items. Choosing the right abstraction depends on how the different use cases throughout the whole program.

In PikeOS, two examples for the varied usage of the single implementation of a linked list data structure are the thread queues used by the scheduler, using the sequence to implement a FIFO strategy, as well as storing the relationship between a task and its children, where the order is insignificant.

An important factor of which abstract representation to choose is how well-supported the specification language feature is, e.g., the degree of automation offered by the prover back-end. In VCC, one fundamental built-in abstraction feature is the representation of (non-primitive) C data structures similar to objects in contrast to the view as simple byte arrays. Two other built-in abstractions are pointer sets, as well as map types.

Not surprisingly, all of these specification features are used throughout the PikeOS specification – for concrete examples for the usage of the set and map abstract data types, see the running example of Section 8.1.1, as well as the specification of PikeOS functionality by Baumann et al. [Bau+11].

Issue B-4: No Language Support for Abstractions. Abstraction in the VCC tool is mainly achieved by using specification ("ghost") state together with built-in abstract data types like maps or sets, where the abstract state is related to the concrete program state with the help of coupling invariants.

One advantage of having a similar notation for the ghost data structures and statements compared to real C is that programmers are already familiar with it. However, at times, such notations are still too close to the source code and better suited alternatives exist – in general, mechanisms to specify abstractions have to be more flexible in order to be able to choose the right abstraction methodology for the task at hand.

Examples. If the right abstract data type has been determined, as explained in the previous example, encoding this abstraction is the next step – we claim that in PikeOS, most data structures, as well as the operations on these structures, would be straight-forward to describe using abstract data types: lists would benefit from an inductively defined abstract type, as we will demonstrate in Section 8.2 to obtain an elegant and concise specification. Another natural candidate for this kind of specification methodology is the PikeOS task tree [KW07].

Approaches to Resolving the Issue. Concerning data abstraction, user-defined abstract data types were already introduced into the VCC methodology, however, there is a large amount of established formalisms, like CASL, that should be taken into further consideration when extending the specification language – in Chapter 8 we explore this option and demonstrate the usage of CASL specifications for a sample verification problem taken from a recent software verification competition.

For control abstraction, many established formalisms exist that could be used for one of the abstraction layers on top of the code, e.g., CSP or abstract state machines. Also, a built-in refinement mechanism is needed to connect the different abstraction levels.

4.3. Local Verification

Deductive software verification tools have improved in recent years to a degree that full functional verification of individual functions written in real-world programming languages is practicable with reasonable effort. This is demonstrated, e.g., by the results of recent verification competitions (e.g., [Kle+11]), where selected software verification problems are solved with limited time resources by teams ranging from verification tool developers to regular tool users.

In the following, we consider as small scale verification the verification of a single function against a *given* informal requirement specification. This includes formalizing the requirement specification in a way that is suitable to the verification methodology at hand, as well as coming up with all auxiliary specifications necessary to verify the function correct w.r.t. its contract.

Issue C-1: Support in Finding Auxiliary Annotations. Finding the right auxiliary annotations for local verification is a complicated task. One issue, finding the right essential annotations at module boundaries, has already been mentioned in Section 4.2 – besides the obvious function contracts, also loop invariants belong to this group. Even if, after some iterations, these essential annotations are appropriate for verifying the function in question, finding auxiliary annotations between modularization points remains to be done.

Until now, there is little support by the tools in finding the right auxiliary annotations: in case of performance problems of the prover, the user can inspect the proof process and investigate which annotation takes up most of the time. In addition, statistics of the underlying SMT solver are reported, e.g., the number of quantifier instantiations so far, to find bottlenecks. In case of too weak essential annotations, VCC produces a counterexample that helps to find the missing or wrong annotations.

If none of these facilities provide the right clue, the user is left with "debugging" the verification state by inserting additional assertions to further split the proof and isolate those parts of annotations that are difficult for the automatic prover.

Example. The general problem to identify the right lemmas, loop invariants, etc. to control the proof strategy for auto-active verification is no different from other (interactive) provers. However, at some points the behavior of the verification system is unexpected and experience in using the verification system is crucial.

A prominent example is the treatment of quantifiers by VCC, respectively Z3: while both tools are able to infer information necessary for quantifier instantiation, the user sometimes has to provide additional guidance as to which terms are relevant for the proof obligation in question. One instance where the inference procedure of VCC fails is the assertion `\exists int i; i == 1`. As a consequence, this assertion cannot be verified by VCC without the user providing another essential annotation in form of a *trigger*, characterizing the terms to match against when instantiating `i` in (the negation of) the sample assertion. In addition, this matching term also has to be given by the user in this example, containing the ground term "1" for quantifier instantiation.

In this simple case, selecting a sufficient trigger is straight-forward – in practice, achieving the right quantifier instantiations in multiple, dependent assertions requires careful considerations. For a more realistic example of trigger selection, see the verification of a simple function given in Section 7.4.2.

Another stumbling block is that not only is the ordering of assertions of importance in finding a proof, but also is the ordering of loop invariant annotations – in more rare cases, even the order of terms within a formula makes a difference. Also, for performance reasons, some information about the proof state is discarded when calling other functions and the user has to explicitly state which information to keep after the function call for the verification of the remaining function.

For further examples of the local verification task, we refer, e.g., to Listing 8.1 on page 150 and Listing 7.8 on page 133.

Issue C-2: Amount of Annotations. Although small scale verification is possible using current verification tools and methodologies with reasonable effort, there is still room for improvements. Most notably, the large amount and high verbosity of annotations is one issue when using auto-active verification tools. In case of VCC, the specification language has been optimized several times to reduce annotation overhead.

Example. The concrete annotation overhead of course strongly depends on the verification task. To give an impression of the dimensions of the effort involved in specifying different types of software, we provide ratios of the amount of annotations or proof script and the corresponding implementation proven correct.

Starting with a simple example of verifying a search operation on a linked list data structure with VCC, as presented in Section 8.1.1 on page 149, the ratio of lines of annotations to lines of implementation is 0.7:1 – however, the annotations are arguably more complex than the implementation and thus comparing the number of tokens would give a different picture.

Regarding a more complex implementation, for the functional verification of parts of the memory manager in PikeOS [Bau+11] the overhead of specification compared to the code compared by lines was 5:1, using VCC as of Feb. 11, 2011. Similarly, Alkassar et al. [Alk+10c] report on a ratio of about 3:1 when comparing annotation and implementation tokens for the baby hypervisor verification.

The magnitude of specification overhead is not specific to VCC and its verification methodology: for the first example of a simple linked list search function, verification with the KeY tool results in a ratio of 1.7:1 lines of specification/implementation. Large-scale verification with the Isabelle framework for the seL4 correctness proof, as already reported in the introduction, clearly emphasizes the issue of large specification and proof script overhead with a ratio in the order of 25:1.

Approaches to Resolving the Issue. One way to reduce the amount of annotations, besides improving the SMT solver, is to identify common specification patterns and create new specification constructs as abbreviation. Another possibility is to choose defaults for specification constructs that cover the most frequent explicit specifications.

In case of C, function declarations are given in separate header files and VCC takes advantage of this by allowing to annotate these declarations with the function contract. This separates the interface specification from the auxiliary specification in the function body but at the same time impairs usability of the verification tool: either the user does not have the contract and the implementation visible side by side when verifying or inspecting the function, or he/she duplicates or moves the contract to the implementation, incurring additional overhead to synchronize the header file with the implementation.

But even with the interface specification taken care of, the problem of a large amount of auxiliary annotations in the function body remains. Again, support of a special verification development environment could help the user to keep track of both implementation and specification. Parts of the annotations irrelevant for the user in understanding the specification could be hidden, given a heuristic that determines the relevancy of an annotation. Information that could be taken into account for such a heuristic is a similarity measure on annotations or formulas (cf. the work by Grebing [Gre12]) – this would allow hiding all but one annotation in a group of similar annotations.

Also, adapting the technique of code refactoring to specifications could be used to reduce the amount and improve readability and maintainability of annotations, e.g., by factoring out often used blocks of specifications in a specification function.

One option to reduce amount of required auxiliary annotations is to shift user interaction towards the proof construction stage, as done in interactive provers. Annotations would only contain the main properties and insights about the implementation (e.g., loop invariants). Proof guidance (e.g., quantifier instantiations) would be done interactively (the information provided by the user can be stored in an explicit proof object for reuse, as in, e.g., the KeY system).

4.4. Handling Software Evolution

While the one-time effort in verifying a large software system is already high, this does not yet include expenditures for re-verification in case of software evolution. To integrate functional software verification into the software development process as part of the software quality control, costs for re-verification have to be reduced, e.g., by re-using as many annotations as possible.

Issue D-1: Change Management. Another issue that arises when verifying an implementation in productive use is that it is constantly evolving, for example, to satisfy changing requirements from users of the software. Also, the verification tool itself can evolve. To cope with these changes, when deciding on the frequency of applying code updates, one has to find a balance between costs for re-verification and the benefits of improvements in the system that might simplify specification and verification.

One possibility is to constantly apply the changes that occur in the production code to the source snapshot that is used as the verification target. While some proofs are unaffected by such changes, it is in general still necessary to adapt annotations and verification proofs. The other possibility is to fix a version of the source code for verification.

Then, to get verified properties for the code used in the actual product, the annotations of the verification target have to be adapted in one big leap to the current implementation at the end of the project.

Example. In the verification of PikeOS within Verisoft XT, one of the major changes was due to the revised memory abstraction of VCC, from the earlier region-based model to the ownership approach still in use in the current version of VCC. At the time of this update in the VCC specification methodology, progress in the verification of PikeOS still allowed to timely translate existing annotations to the new specification language, rather than working with annotations for two different methodologies at the same time.

Regarding the implementation of PikeOS, we benefited from the fact that we were working with production code deployed in real-world safety-critical systems – implying that the interface of PikeOS and thus the requirement specification remained stable. Changes in the implementation of the production version of PikeOS were transferred iteratively to the verification target in the beginning of Verisoft XT– at a later point, we decided to fix the implementation of the verification target and no further updates were applied. To support change management for these code updates, we used a test harness for regression testing to check validity of proofs completed for earlier program versions.

Part II.

Improving Deductive Verification for Real-World Application

Introduction

In the first part of this thesis we have demonstrated how a state-of-the-art annotation based deductive verification tool is used to verify a complex, concurrent software system by means of a simulation proof. We have also shown that a correctness proof for this type and size of implementation is far from being trivial and we have given some reasons why this is the case, based on our experience in specifying and verifying the PikeOS microkernel within the Verisoft XT project.

The subsequent chapters of this thesis shall contribute to improve upon the productivity of working with an annotation-based program verification system like VCC: on the one hand, this concerns interaction of the user with the verification tool. Here, we envisage qualities like a simple but expressive specification language, or adequate feedback for failed verification attempts. In this regard, in the following sections we will (a) address the issue of missing language support for abstractions (Issue B-4 on page 68) by proposing to use existing specification constructs to give succinct specifications for data structures in Chapter 8, (b) simplify finding the right auxiliary annotations and help with entangled specifications (Issue B-1 on page 65) by providing early feedback for (partial) specifications in Chapter 7 and (c) describe the different roles of annotations in auto-active tools in Chapter 5 the user needs to be aware of to be able to state auxiliary annotations in a systematic fashion – this section is also concerned with the issue of a missing syntactic distinction between the different kinds of annotations in VCC described in Issue A-2 on page 63.

On the other hand, productivity with the tool in the long term also depends on the stability of the “interface” to the verification system consisting of the specification language and verification methodology: In the case of a change in the system, the users not only have to familiarize themselves with the new verification system, but also any change in the implementation and axiomatization of the verification tool may affect previously completed proofs, which then have to be repeated. However, such changes are almost unavoidable during the development of a verification tool in order to fix errors in the tool – due to the complexity of the implementation and specification language, generating the right proof obligations that imply correctness of the concurrent C code w.r.t. its specification is far from trivial.

One option to reduce the number of bug-fixes needed for the verification tool is to use software testing. Assessing the quality of corresponding tests is discussed in Chapter 6 – test suites improved according to our quality metric thus indirectly reduce the amount of work needed in change management for the verification system (Issue D-1 on page 71).

In the following we will give a short summary of the four topics addressing the issues mentioned above.

Different Roles of Annotations in Auto-active Tools We first examine the implications of the auto-active user interaction paradigm on the completeness of such a system and hence also on the usability of the verification tool. Based on these considerations, we arrive at the definition of *annotation completeness* – this property states that there is always a set of annotations sufficient to give rise to a correctness proof (given that the program is indeed correct w.r.t. its specification), for a verification system that is complete according to this definition.

Furthermore, this allows us to give a more precise terminology for the different types of auxiliary annotations: we can distinguish between annotations needed only for efficiency reasons and *essential* annotations needed for the existence of a proof. We claim that the user has to be aware of this distinction in order to provide the verification system with the right kind of annotations. Also, any technique supposed to optimize specifications (e.g., by reducing the number or complexity of annotations) may exploit this distinction to focus only on a certain type of annotation.

Testing Axiomatizations of Deductive Verification Systems Deductive verification systems usually consist of both an implementation part, as well as a set of rules or axioms of the calculus used – soundness of the overall system necessarily depends on the correctness of each component together with their proper interaction.

One option to reduce the amount of bugs in the overall system and which can be integrated easily into the software development process is to use software testing techniques. To ensure that sufficiently many as well as the right kind of test cases are written, it is important to assess the quality of the test suite. While for the implementation part established quality metrics like statement or branch coverage exist, a corresponding metric for test coverage of the axiomatization respectively rule base is missing. We will introduce the notion of *axiomatization coverage* in Chapter 6, together with two case studies evaluating the adequacy of this coverage definition.

Early Feedback for (Partial) Specifications One of the factors that contributes to the complexity in finding appropriate auxiliary annotations for auto-active verification tools is that meaningful feedback of the tools is only possible after essential auxiliary annotations have been provided by the user. In Chapter 7 we address this issue by adding a bounded model checking tool for whole-system analysis in the verification tool-chain.

Using bounded model checking, we are able to check for single annotations whether (bounded) program executions adhere to the specification. In case an annotation is violated, a counterexample in terms of a concrete program execution is returned – this information can then be used by the verification engineer to modify the program or specification, enabling an improved mode of operation of the verification tool we call counterexample-guided *manual* annotation refinement (CEGMAR).

Succinct Specifications for Data Structures Verbosity of specifications is a common issue in verification of real-world programs. To counter this problem inherent to the specification of complex systems and properties, two options that are used in practice are changes to the verification methodology, as well as introduction of new features of the specification language. For common specification patterns, instead of constructing them from basic annotation constructs, the specification language should offer support with appropriate mechanisms in order to be able to give concise annotations.

For instance, data structures used in the implementation of a system can often be easily described in terms of their abstract data types counterpart, yielding an elegant specification with an "interface" that is well understood. We thus propose to extend the annotation language of VCC by elements taken from the established Common Algebraic Specification Language (CASL) [Ast+02] in Section 8.2, making use of the experience in designing a language that the authors claim "is based on a critical selection of known constructs" to get "an expressive, simple, pragmatic language" [CASL]. We demonstrate at an example taken from 1st Verified Software Competition [Kle+11] how a specification based on abstract data types compares to the contracts given in the regular VCC annotation language.

5. The Auto-Active Verification Paradigm

Contents

5.1. Introduction

It is widely recognized that human input is indispensable in deductive verification of real-world code. Verification engineers have to guide the proof search and provide information reflecting their insight into the workings of the program.

In this chapter, we clarify the different purposes that annotations can serve and show why a certain class of annotations that are not program requirements is currently indispensable for proof construction.

The problems with annotation-based verification arise as the lines between (a) requirement specification, (b) auxiliary information needed for proof construction, and (c) information for proof-search guidance get blurred. Even worse, related specification parts get dispersed and different levels of abstraction intermixed.

We discuss in Section 5.2 the different purposes that annotations can serve, based on a clarification of what the notion of completeness means in this framework. We show why a certain class of annotations that are not program requirements is currently indispensable for proof construction. Users are often surprised that they cannot omit certain non-requirement annotations even for the simplest (sub-)problems. We plead that they must be better educated about the inner workings of a verification system and what kinds of annotations are indispensable in which situations. This is in conflict with the desire to enable the user to work with the verification system as a “black box”, which is generally seen as an important feature of the verifying compiler paradigm.

The problems we discuss are not inherent to annotation-based verification (nor the verifying compiler approach), but rather due to the currently implemented design decisions of such verification systems. All the issues mentioned in this chapter can be overcome by extending specification languages and methodologies. The general, central idea of annotating programs at source-code level – using a language whose syntax is closely related to the programming language – is not the source of the problems described in this chapter.

5.1.1. The Possible Outcomes of Invoking an Annotation-based Verification Tool

Recall from the introduction to VCC in Section 2.4.1 that the tool works using an internal two-stage process: First, in the verification condition generation (VCG) stage, the annotated C code is translated to the intermediate language Boogie before being converted into a proof obligation as a set of first-order logic formulas. In the second stage, the resulting formulas are given to an automatic theorem prover (TP) respectively SMT solver (together with a background theory).

In practice, where the limitations of resources are relevant, the possible outcomes of a verification attempt using this two-stage annotation-based verification system are:[1]

1. The formulas generated by the VCG are valid, and the TP has found a proof for that. This outcome entails that the original program has the specified properties.

2. Some generated formula is not valid, and the TP has found a counter-example. This can mean two things: (a) The program is not correct w.r.t. its specification, i.e., there is an error in either the program code or the specification. (b) The program satisfies the specification, but some loop invariant or other auxiliary annotation is missing or not strong enough and, as a consequence, some generated verification condition is not a valid formula. We will discuss this distinction in more detail in Section 5.2.1.

3. The TP runs out of resources (time or space). This can mean two things: (a) The generated formula is valid and the program is correct (as in Case 1 above), but the TP could not find a proof in the allotted time/space. (b) The formula is not valid (as in Case 2 above), but the TP could not find a counter-example. The non-validity can, again, be due either to the program being incorrect or to some auxiliary annotation being not strong enough.

In Case 1 above, the invocation of the verification system was successful – a desired but rare case in practice. Cases 2 and 3 are much more common, and the user has to analyze the problem. If the user finds (using the potential counter-example) that the program indeed does not satisfy the specification, the error has to be corrected. Otherwise, if the user finds that the program satisfies the specification, then new auxiliary annotations (stronger invariants, helpful lemmas, etc.) have to be added. This process is repeated until the program can be verified.

[1]We assume that the programs to be verified are of reasonable size such that only the theorem proving stage can run out of resources and not the VCG stage.

5.2. Distinguishing Different Kinds of Annotations

5.2.1. Annotations and their Properties

Preliminaries In the following we assume as given a programming language, and an annotation language for expressing specifications. Which annotations are possible depends on the particular language; typical annotations are for example invariants, pre-/postcondition pairs, and assertions of various kinds. In order to easily relate alternative potential annotations for the same program, we take the view that annotations are disjoint from regular program statements. On the other hand, each annotation has an *intended context* (statement, method, class, etc.). We assume that we only deal with combinations of programs and annotations without context mismatches, i.e., annotations are compatible with the programs to which they are added. The context an annotation refers to must actually exist in the program, and the symbols used in the annotation must be defined for that context.

Definition 1 (Combination of program and annotations)**.** *If P is a program and A is a set of annotations compatible with P, then we call the pair $P +_{<} A$ the* combination *of the two. The parameter $<$ fixes the order of annotations if several of them have the same intended context. We will omit the ordering whenever it is irrelevant or clear from the context and simply write $P+A$.*

Definition 2 (Annotation satisfaction)**.** *We assume that there is a definition of when a program P satisfies a specification REQ, denoted by $\models P+REQ$.*

Definition 3 (Strength of annotations)**.** *An annotation A is* (logically) stronger *than an annotation B, in symbols $A \Rightarrow B$, if $\models P+B$ holds for all programs P with $\models P+A$.*

Different Purposes of Program Annotations Annotations can serve distinctively different purposes, though sometimes several ones simultaneously. The following classification of annotations is neither syntactic nor semantic, but concerns rather the pragmatics of their use and the intentions of their author.

Requirement Annotations. Requirement annotations constitute the specification of the program. They assure the behavior of the program (module) towards its environment. They are the reason for performing verification. Typical requirement annotations are pre- and postconditions, class invariants, or resource consumption limits. They are visible externally and cannot be changed easily.

Auxiliary Annotations. Auxiliary annotations are used to guide the proof search. They are usually not part of program requirements. As long as they satisfy their purpose, auxiliary annotations can be changed anytime without notice. We further distinguish two subclasses of auxiliary annotations:

(a) The first subclass is necessary merely for efficiency reasons. It encompasses lemmas, intermediate assertions, quantifier instantiation triggers, and the like. These annotations are not necessary for completeness. They can always be made obsolete by increasing the space/time available for proof search or by advances in SMT prover technology. Another purpose of annotations from this subclass is to inspect the proof state. For this, the user temporarily adds auxiliary annotations to get information about implicit "knowledge" of the proof system at particular points in the proof search – in order to eventually come up with the right auxiliary annotations needed to complete the proof (as defined in the following in Def. 7).

(b) The other subclass of auxiliary annotations are essential annotations. Getting them right is essential for completeness, the very existence of a correctness proof. The most prominent essential annotations are loop invariants. Further auxiliary annotations that can be essential are data-structure invariants and abstractions, ownership annotations, and framing conditions.

Monotonicity of Auxiliary Annotations A very desirable property of annotation satisfaction is monotonicity. Adding auxiliary annotations should strictly increase the strength of the specification, i.e., $\vDash$ should be monotonic w.r.t. adding annotations.

Definition 4 (Monotonicity of $\vDash$)**.** *$\vDash$ is* monotonic *w.r.t. adding annotations iff, for all programs P and all specifications REQ and AUX the following holds:*

$$\textit{if} \ \vDash P{+}(REQ \cup AUX) \ \textit{then} \ \vDash P{+}REQ \ .$$

In reality, monotonicity of $\vDash$ w.r.t. adding auxiliary annotations is not given unless we make some restrictions. One concerns `assume` annotations that add an unchecked assumption to the following proof (and thus can make a specification weaker). Since all proved properties only hold modulo these assumptions, `assume` annotations are a correctness risk. For these reasons we always classify `assume` annotations as part of the requirement and never as auxiliary.

Similarly, adding a formula to a precondition, and thus weakening it, violates the condition of Definition 4 and is not an acceptable way of adding auxiliary annotations.

5.2.2. Annotations and Existence of Proofs

To separate the annotations that are essential for the existence of a proof and the ones that are needed only for supporting proof search, one needs a clear understanding of the notion of completeness. In this section, we discuss what completeness means in the framework of annotation-based verification systems and give formal definitions.

Completeness and Relative Completeness

The classical notion of completeness for deduction systems can be adapted to annotation-based verification systems as follows:

Definition 5 (Completeness)**.** *Let S be a verification calculus or system. S is* complete *if, for any program P satisfying its requirement specification REQ, this fact can be proved using the calculus from a fixed set of axioms Th_S. In symbols:*

$$\textit{if} \ \vDash P{+}REQ \ \textit{then} \ Th_S \vdash_S P{+}REQ$$

The semantics of the programming language (used for P) and the annotation language (used for REQ) are encoded in the calculus rules $\vdash_S$ and in the background theory Th_S. The restriction of resources (time and space) of real-world systems is usually not considered for the notion of completeness.

Note also the difference between $\vDash$ and $\vdash$: Fewer annotations are easier to satisfy by the program ($\vDash$), while more annotations may make it easier to find a proof ($\vdash$).

Since all non-trivial properties of programs are undecidable (Rice's Theorem), all program verification systems are necessarily incomplete in the sense of Definition 5. Instead, the notion of *relative completeness* is used, i.e., completeness in the sense that the system or calculus would be complete if it had an oracle for the validity of formulas about arithmetic [Coo78]. This can be formalized as follows:

Definition 6 (Relative completeness)**.** *A verification system S, consisting of $\vdash_S$ and Th_S, is* relatively complete *(w.r.t. arithmetics) if, for each program P and specification REQ with*

$$\vDash P{+}REQ \quad ,$$

there is a set $Arith$ of valid arithmetical formulas such that

$$Th_S \cup Arith \vdash_S P{+}REQ \quad .$$

Luckily, undecidability of first-order arithmetics is usually not an obstacle for verification in practice. Experience shows that the axiomatization Th together with the calculus rules of the theorem prover approximate arithmetics well enough and that the valid arithmetic formulas occurring in practice can be derived (which does not imply that finding a derivation is easy or possible automatically but only that a derivation exists). One has to keep in mind, that the distinction between completeness and relative completeness exists, even if the restriction to relative completeness is not a real limitation in practice.

Theoretical Completeness Arguments

Relatively complete calculi exist for many program logics. Harel gives one for first-order Dynamic Logic in [Har79]. Less-known is the fact that the presence of auxiliary annotations, such as loop invariants, is not a prerequisite for relative completeness.

Harel conducts his relative completeness proof by showing that program logics are no more expressive than first-order arithmetics. That is, for every program there is a first-order arithmetics formula that encodes the same relation between states that the program encodes. The less-known fact is that it is possible to effectively compute such a formula without further input. In fact, Harel's proof contains a simple algorithm that for any Dynamic Logic formula ϕ effectively computes an equivalent purely first-order arithmetics formula ϕ_A [Har79, Theorem 3.2]. This construction gives along the way a means to automatically compute a strongest invariant of any loop.

The algorithm is based on Gödelization, i.e., encoding a finite sequence of domain elements into one element. The generated invariant formula asserts the existence of a number encoding a sequence of states corresponding to the forthcoming computation sequence of the loop until it terminates.

Thus, theoretically speaking, the strongest loop invariant for any given loop and so the verification conditions for any given piece of code can be easily computed. That is not a contradiction to general undecidability of program verification, since one undecidable problem (program verification) gets transformed into another undecidable problem (deciding first-order logic with arithmetics). Still, assuming a theoretical standpoint, one can conclude that no auxiliary annotations are really needed because all information contained in annotations can easily be computed by the VCG.

In Practice: Annotation Completeness

The theoretical fact that all necessary annotations can "easily" be constructed (see above) is in practice a red herring because the constructed annotations use Gödelization and, thus, complex arithmetics. Proof obligations generated from such annotations would be impossible to discharge by existing theorem provers. For practical purposes one needs instead annotations containing all the necessary information in a clear and direct manner and not obscured by Gödelization.

Therefore, in contrast to theory, all of today's deductive verification systems presuppose certain types of additional, non-requirement annotations to be given by the user. It is neither given nor expected that an annotation-based verification system is relatively complete in the sense of Definition 6. In practice, completeness of a verification system means that if the program is correct w.r.t. its *given* requirement specification REQ, then some auxiliary specification AUX *exists* allowing to prove this.

Definition 7 (Annotation completeness)**.** *A verification system* $(\vdash_S, Th_S)$ *is* annotation complete *if, for each program* P *and specification* REQ *with*

$$\models P{+}REQ \quad ,$$

there is (a) a set AUX *of annotations (not containing any* `assume` *clauses), (b) an order* $<$ *on the annotations, and (c) a set* $Arith$ *of valid arithmetical formulas such that*

$$Th_S \cup Arith \vdash_S P{+_<}(REQ \cup AUX) \quad .$$

The completeness of the whole verification process depends on completeness of the components of the toolchain. As already described, the toolchain usually consists of a VCG stage and an automated theorem proving or SMT back-end. The VCG must be able to generate valid formulas provided the auxiliary annotations are sufficiently strong, i.e,

$$\text{if } \models P{+}REQ \text{ then } Th \models_{FOL} VCG(P{+}(REQ \cup AUX))$$

for some AUX. Then the TP, in its turn, must be able to prove these valid formulas:

$$Th \vdash VCG(P{+}(REQ \cup AUX)) \quad .$$

The users, who serve as an oracle for finding auxiliary annotations that are strong enough to prove a given program correct are not relevant for the completeness as long as they are considered to be omniscient and always find the required annotation (provided it exists). In practice, of course, users are not omniscient. They may very well fail to find the required auxiliary annotation, which may lead to a failure in the verification process even if the verification system is complete.

Note that, if one annotation-complete system S is stronger than another annotation-complete system S' because it can automatically derive additional annotations (it may, e.g., include a generator for loop invariants), then life is easier for the user of S; proofs will be found more often using S and with less effort (less auxiliary annotations). Nevertheless, both systems S and S' are annotation-complete; there are no different degrees of annotation completeness.

Essential and Non-essential Annotations

When a verification system is used that is annotation complete (Def. 7) but not relatively complete (Def. 6), i.e., any annotation-based verification system, then there are *essential* auxiliary annotations that cannot be omitted without losing the existence of a proof. Besides such essential annotations there are *non-essential* annotations that are not needed for the existence of a proof but for finding it more easily.

Definition 8 (Essential annotation)**.** *Given a verification system* $(\vdash_S, Th_S)$*, a program* P*, a specification* REQ *with* $\vDash P{+}REQ$*, and a set* AUX *of annotations and an order* $<$ *with*

$$Th_S \cup Arith \vdash_S P{+_{<}}(REQ \cup AUX)$$

for some set $Arith$ *of valid arithmetical formulas.*

A subset $AUX_{ess} \subset AUX$ *is* essential *if*

$$Th_S \cup Arith \nvdash_S P{+_{<}}(REQ \cup (AUX \setminus AUX_{ess})) \quad .$$

Otherwise, it is non-essential*.*

The notion of essential annotations (Def. 8) has some awkward properties, which make it difficult to recognize essential annotations in practice. It is possible that some subsets $A, A' \subset AUX$ are both essential but $A \cup A'$ is not. This happens frequently if, for example, A is needed for the proof of A' and A' is needed for the proof of A. Also, there is in general no single minimal set of essential annotations. In fact there may be completely different sets of essential auxiliary annotations for proving the same requirement that are both minimal but disjoint.

Besides whether an annotation may be omitted or not, one may also be interested in whether it can be replaced by a weaker annotation.

Definition 9 (Strongly essential annotation)**.** *A set* $AUX_{ess} \subset AUX$ *is* strongly essential *if*

$$Th_S \cup Arith \nvdash_S P{+}REQ \cup ((AUX \setminus AUX_{ess}) \cup AUX')$$

for all AUX' *that are weaker than* AUX_{ess}*, i.e.,* $AUX_{ess} \Rightarrow AUX'$*.*

While an auxiliary annotation that is strongly essential cannot be replaced by a weaker annotation, it may well be possible to replace it by an equivalent annotation that is "simpler" in some practical way not covered by Definition 9 (e.g., easier to understand for human users).

Note that even with the notion of strongly essential annotations, there is in general no single minimal set of auxiliary annotations (e.g., for a requirement specification with postcondition $P \vee Q$, where P and Q are independent, and a set AUX of annotations that suffices to prove both P and Q, there might be two sets of strongly essential annotations AUX_P and AUX_Q sufficient to prove only P respectively only Q).

It is important for a user to know which annotations are essential because during the verification process many auxiliary annotations are added. And as too many annotations clutter the program and make it harder to find a proof, users often remove unneeded annotations. This carries the danger that simple but essential annotations get removed by accident, which – as experience shows – leads to hard to solve problems in the search for a set of annotations with which a proof can be constructed. Thus, to understand which annotations may be essential, users have to possess a certain knowledge about the inner workings of a verification system. As further discussed in Section 5.2.4, we also suggest enriching the annotation languages with a syntactical way (e.g., a keyword) to distinguish between the two kinds of annotations.

5.2.3. Possible Failures in Authoring Annotations

In the following, we use a concrete example to illustrate the three different ways in which authoring annotations may fail.

Annotations and Program Code Can Be In Conflict. A program P and an annotation $SPEC$ are in conflict if the program does not fulfill the specification: $\nvDash P{+}SPEC$.

Consider the code in Listing 5.1 together with the requirement to compute the minimum of a given array of length `size`. The precondition of the method states that `array` points to a C array in memory with positive length `size`, which is not modified outside the current thread (the latter enables sequential reasoning). The post-condition of the method states that the result of the method is (a) less than or equal to all elements and (b) contained in the array. We assume in the following that this is the right set of requirement annotations.

One possible error that could occur in the program is that the variable `min` has never been initialized (line labeled (A) missing). The resulting program is legal C code, but depending on the random initial value of `min` and the contents of the array, may fail to compute the minimum, and it does not satisfy the annotations.

For this conflict, the VCC system is able to provide a counter-example. It demonstrates that the second loop invariant does not hold when the loop is entered. The variable assignment returned as counter-example is: `size = 1`, `min = 0`, `array[0] = 1`.

```
#define uint unsigned int

int min(int *array, uint size)
  _(requires size > 0)
  _(requires \mutable_array(array, size))
  _(ensures  \forall uint i; 0<=i && i<size ==>
                      result <= array[i])
  _(ensures  \exists uint i; 0<=i && i<size &&
                      result == array[i])
{
    uint i;
    int min;
    min = array[0];                                      // * (A) *
    for (i = 0; i < size; i++)
      _(invariant \forall uint j; 0 <= j && j < i ==>
                            array[j] >= min)
      _(invariant \exists uint j; 0 <= j && j < size &&
                            min == array[j])            // * (B) *
    { if (array[i] < min) min = array[i]; }
    return min;
}
```

Listing 5.1: Annotated implementation computing the smallest element of an array by simple iteration

Annotations Can Be Too Weak. An auxiliary annotation AUX is too weak if the program is correct w.r.t. the specification, but this cannot be shown, i.e., $\models P{+}REQ \cup AUX$. There are now two cases to distinguish:

1. The VCG produces valid verification conditions, i.e.,

$$\mathit{Th} \models_{FOL} VCG(P{+}(REQ \cup AUX)) \ ,$$

and there is a proof for this, i.e.,

$$\mathit{Th} \vdash VCG(P{+}(REQ \cup AUX)) \ ,$$

but the TP stage runs out of resources before finding a proof.

2. Something essential is missing from AUX and at least one of the verification conditions generated by the VCG is invalid:

$$\mathit{Th} \not\models_{FOL} VCG(P{+}(REQ \cup AUX)) \ ,$$

and (because of soundness) no proof exists, that is:

$$\mathit{Th} \nvdash VCG(P{+}(REQ \cup AUX)) \ .$$

In Case (1), no counter-example is available and the user has limited recourse – to assist the user, VCC provides tools for inspecting the duration of proof attempts for single proof obligations as well as identifying axioms that are "costly" for the prover to instantiate, leading to an inefficient proof search. In Case (2), a counter-example for the validity of the verification condition may be constructed. We give an example for this latter case.

Assume that the second loop-invariant has been forgotten (label (B) in the program in Listing 5.1). Without that invariant, the system cannot verify the second post-condition. The generated counter-example is still the same as above, but this time it shows that the loop invariants (after the loop terminates) do not logically entail the post-condition.

Annotations Can Be Inadequate. An annotation is inadequate when it does not mean what its author thinks it means. Verification of inadequate annotations will thus not have the expected impact in the real world. By its very nature, user input cannot easily be verified or tested for adequacy. But, apart from many systematic approaches for elicitation of requirements (which we will not cover here), there are a number of ways in which verification technology can assist its user to formulate meaningful specifications.

First, the builders of verification systems can work on formalisms that do not make it unnecessarily hard for the users to express their exact intentions. Second, the verification systems can produce a proof or a trace to justify the result. Inspection of the proof is a very effective – if costly – measure to combat misunderstandings in the meaning of the proof obligation. There are reports that users of verification systems monitor the prover running time to detect verification based on inadvertently inconsistent specifications (a particular case of inadequacy). In addition, VCC can check for inconsistencies in the specification by trying to prove *false* at the different execution branches of the program – this of course can only give an indication whether the specification is consistent or not.

Third, a new class of sanity checks based on mutation has been developed lately for automated program verification with model checking [Kup06]. After a successful verification attempt, the query (the program or the specification) is mutated and the deduction is repeated. If verification succeeds again, then the mutated part of the query probably plays no role in determining the outcome. This indicates a problem with the query.

5.2.4. Improving the Annotation Languages and Methodologies

Annotation-based verification systems are currently not designed for completeness in the sense that theorem provers are (Def. 6). They are designed for completeness in a different sense (Def. 7), requiring the user as an oracle to provide sufficient auxiliary annotations in the form of, e.g., loop invariants or assertions.

Theoretically the user could always give auxiliary annotations of maximal strength (i.e., logically entailing all other possible annotations), but this is not feasible in practice. Instead, one is interested in a weak set of auxiliary annotations that is still sufficient. Consequently, it is extremely important for the user to have knowledge about which kind of annotations are essential for the given VCG – even in cases where the requirement to be verified is comparatively simple. Without that knowledge they may continue to add the wrong annotations. It is therefore essential to provide user documentation on what kind of auxiliary annotations are needed by a verification system.

Moreover, requirement and auxiliary annotations must be syntactically distinguished. That makes specifications clearer and easier to read and understand. In certification processes it is indispensable to have a very clear understanding of which annotations form the requirement specification that has been verified.

It is preferable to keep the two in separate files: for instance, requirement annotations in header files and auxiliary annotations in C source files. If no such separation is possible, keywords (in the style of visibility modifiers `public` and `private`) should be used.

6. Improving Trust in Verification Systems

Contents

Correctness of a verified program or computer system w.r.t. its specification always rests upon the soundness of the verification system used to conduct the proofs. With the objective of the Verisoft projects of pervasive verification in mind, the logical conclusion is to treat the verification tools with the same analytical rigor as every other component involved in the system. Proving soundness of the proof calculi and faultless implementation of, e.g., the rule application mechanism is one option to lend trust to the overall correctness argument. Another benefit of a correctness proof of the verification systems used is that the proof may serve as evidence for tool qualification in a certification process, such as the DO-178C [RE12] in avionics.

Besides these obvious advantages, striving for a formal correctness proof also has its drawbacks, as noted by Beckert and Klebanov [BK06]: One of the issues identified was the effort needed to produce a formal proof, which might better be spent towards other tasks (at least as far as research projects are concerned). In addition, conclusiveness of such a proof is limited if no official, formal specification of the target programming language semantics is given – proving the verification system to work correctly according to a faulty language definition does not provide the intended evidence for tool qualification.

Developing a verification system for a real programming language that meets the demands placed by the system and properties to be verified is an iterative process. One example illustrating the obstacles encountered during tool development is the VCC tool within Verisoft XT. In this case, the tool has been written as part of the project, evolving with the requirements posed by both the Hyper-V and PikeOS case studies. New specification techniques, as well as performance issues in verification necessitated changes to the VCC tool. Note that this tool evolution and the issues involved are not limited to the particular setup of a newly developed verification tool and a complex case study, but present in most systems used for research.

From our experience in Verisoft XT it is important to speed up this iterative improvement process in order to get a stable tool version as soon as possible in a verification project. Later changes in the verification tool increase the effort needed to make the required modifications to existing program specifications.

The not inconsiderable frequency of changes to the verification system, which is mostly unavoidable for a project of the scale like Verisoft XT, renders time-consuming correctness proofs for the verification tools infeasible. Instead, we propose a different technique based on software testing to increase trust in the verification system and reduce the probability to introduce regression bugs. Reducing the amount of regressions then helps to lessen the need to introduce changes to the verification tool.

While standard software testing approaches are applicable to verification systems, they only focus on the implementation part of the tool – for VCC, this amounts to the translation from the source language C to the "compiled" target language Boogie. Equally important to the correctness of VCC is the background axiomatization (or prelude) as presented in Section 2.4.1, containing, e.g., the declarative description of parts of the C programming language semantics.

Based on earlier work by Wagner [Wag09], our proposed technique focuses on this axiomatization part of the verification systems (see Section 2.4.1) – the presented *axiomatization coverage* criterion provides a metric to assess and improve the quality of test suites for the verification tools. Using the same idea, we are also able to generate additional regression test cases from existing tests. If used with test cases from other verification systems (where applicable), this testing technique provides the cross-validation results as suggested by Beckert and Klebanov [BK06].

In the following, we extend on the work of Wagner [Wag09] by first providing a more precise definition of the axiomatization coverage and the notion of *strongly covered* axioms. The case studies performed in previous work are extended to the KeY system, (a) acquiring new data that allows for more detailed analyses, and (b) providing evidence that axiomatization coverage is a sensible coverage metric. The structure and content of the following sections is based on our previously published paper at TAP'2013 [BBW13a].

6.1. Targets of Evaluation

Our approach of testing deductive program verification tools presupposes a certain interaction mode between the user and the tool similar to auto-active verification found in VCC: the user has to be able to provide sufficient annotations so that the prover is able to establish the proof without further interaction. Many software verification tools besides VCC fulfill this criterion, such as other auto-active tools including the Jessie plug-in of Frama-C [MM13] (if automatic provers are used as back-end), or Spec#.

Verification tools allowing user interaction during the proof construction stage, like the KeY tool, are also suitable for our testing technique: in many cases, these tools can be used in an auto-active manner without relying on user input during proof construction.

Hence, for the rest of this chapter, we restrict all test cases to be provable without interaction. This also allows us to treat both case studies made with VCC and KeY similarly.

Another prerequisite for our technique to be applicable at all is that the domain knowledge (e.g., the semantics of the target programming language) is encoded declaratively as an axiomatization or set of calculus rules. This is the case for both VCC and KeY– in contrast, static program analysis tools like LLBMC implicitly contain the definition of specification and programming language semantics as part of their implementation and are not amenable for the proposed testing method.

6.2. Test Cases for Program Verification Systems

The character of test cases for a program verification system depends both on the properties to be tested and which part of the system is tested, together with the level of granularity the components are examined.

For verification tools with a modular implementation like VCC, the different stages, from the verification condition generator (VCG) up to the SMT solver, are easily discernible and can in principle be validated in isolation. Using software testing approaches, each of these components could in a first step be validated with the help of system tests.

The difficulty of writing these system tests varies with the component tested: for the particular SMT solver of the toolchain, on the one hand, there are many comparable tools with a similar input language, facilitating reuse of existing test cases. On the other hand, providing an oracle to deliver a verdict for a test run is comparatively easy due to the small range of possible outcomes of the SMT tool run – one option is to use the result of another SMT solver as oracle. In contrast, specifying when an output of the VCG stage of the verification tool passes the test is more elaborate and providing an appropriate oracle even more so.

The approach described here for testing verification systems avoids this difficulty by testing the whole verification toolchain with the help of system tests. In doing so, we lose some precision to locate errors in the tool under test due to the interplay of the different components. However, this testing technique is also applicable to verification tools with a more monolithic architecture, like the KeY system.

A system test consists of an annotated program $P{+}SPEC$, as defined in Chapter 5, as well as an oracle in the form of the expected output of the verification tool provided by the test author. As we require the tests to run without further user interaction, we assume that for all tests the set $SPEC$ to contain sufficient auxiliary annotations to allow a conclusive result by the verification tool.

Our testing approach is restricted to reveal issues in functional correctness of the verification tool – other qualities like efficiency, resource consumption or usability, while important properties for the application of the tool, are not examined.

6.3. Testing Different Properties

The main deficiencies of verification systems we are interested in w.r.t. testing can be roughly divided into the following four categories: (a) unsoundness: the system incorrectly verifies a program that does not meet its specification as correct (b) (annotation) incompleteness: the system fails to find a proof for the correctness of a program, regardless of the set of annotations provided (c) bad performance: the system uses more resources than expected or requires additional or more complex *non-essential* auxiliary annotations than anticipated to give a definite result (d) abnormal termination: the system crashes.

In the following, we are mainly concerned with revealing failures from category (a) and (b) through testing. In addition, in practice, ensuring a good performance of the verification tool is also important. Completing the test cases for category (a) and (b) in reasonable time ensures that at least for some programs and corresponding annotations performance of the verification tool is adequate. Whether in practice the user is able to provide a set of annotations that allows to *quickly* get definite results is out of the scope of this work. Similar, while abnormal termination of software is usually a serious concern, a bug that makes the verification system crash is in general less serious than a completeness failure because it cannot be confused with bugs in the program to be verified (the verification target).

	Intended		
Observed	provable	not provable	timeout
proved	—	unsoundness	unsoundness or positive performance anomaly
not provable	incompleteness or performance failure w.r.t. required annotations	—	incompleteness or positive performance anomaly
timeout	incompleteness or performance failure w.r.t. required annotations or resources	unsoundness or performance failure w.r.t. resources	—

Table 6.1.: The different results of tests for auto-active verification systems and the failures they indicate.

Table 6.1 summarizes the different test results and the failures they indicate. The possibility of abnormal termination exists in all cases and is not included in the table. If a failure type is not listed in a cell, the verification tool may nonetheless show this deficit for the combination of intended and observed result – e.g., a test case intended to be provable may have been proved by the tool through an inconsistency in its axiomatization.

The coarse categorization of intended and observed results can be refined to be able to draw further conclusions from a test output. In the case of a test that is intended to be provable, the author of the test case may have certain expectations on how the verification system conducts the proof, in addition to the fact that it is indeed provable. Therefore, proof statistics could be collected to be matched against these expectations. Compared with earlier test runs, evaluation of the proof statistics might detect anomalies in comparison to earlier test runs (e.g., an unusually short proof might indicate unsoundness in the axiomatization). The same holds for test cases intended not to be provable that are proven by the system under test: here, our technique to compute axiomatization coverage described further could be used to collect information about the source of the unsoundness. Also, the "timeout" column in Table 6.1 could be split into tests that are intended to be provable and those that are not, allowing for a more precise characterization of the failures a test result indicates.

How one can test for different kinds of failures, and thus, correctness properties, is discussed in the following subsections.

Testing Soundness For our considerations, the most important property of verification tools is soundness. This means that whenever the output for a verification problem P+$SPEC$ is "proved", then the program P indeed satisfies the specification $SPEC$.

To reveal a soundness bug, a test case must consist of a program P and a specification $SPEC$ such that P does *not* satisfy $SPEC$. The correct answer for such a verification problem is "not provable" or "timeout", while "provable" indicates a soundness failure. In practice, though, the result "timeout" is unsatisfactory as it does not allow distinguishing an unsound but inefficient from a sound verification system – as a consequence, the test author should aim for test cases for which annotations can be provided that are strong enough for the prover to be able to give a definite answer.

Programs that satisfy their specification cannot reveal soundness bugs – at least not directly. The exception are cases where the expected answer is "timeout" but the system answers "proved". Such an anomaly – that needs to be investigated by the developer – can either stem from an unexpected good performance or from a short but incorrect proof (i.e., a soundness problem).

Testing Completeness We have established in Chapter 5 the concept of *annotation completeness* to characterize the notion of completeness adequate for annotation-based verification tools: If a program is correct w.r.t. its *given* requirement specification REQ, then some auxiliary specification AUX or other required user input *exists* allowing to prove this.

Hence, to reveal a completeness problem, a test case must consist of a program P with annotations $REQ \cup AUX$ such that (a) P satisfies $REQ \cup AUX$ and (b) the annotations are strong enough to prove this, i.e., the expected output is "proved".

If the observed output is "not provable", then a completeness failure is revealed. In that case, a proof may exist using a different (stronger) set AUX' of auxiliary annotations. That is, the system may or may not be annotation complete. The failed test only shows that the expectation that a proof can be found using the annotation set AUX does not hold. The situation is similar with an observed output "timeout". In that case, the system may just be slower than expected or, worse, there may be no proof using AUX or, worst of all, there may be no proof at all for any annotation set AUX'. For both kinds of incorrect output ("not provable" and "timeout") the developer has to further investigate what kind of failure occurred.

One may consider incompleteness of a verification tool to be harmless in the sense that it is noticeable: the user does not achieve the desired goal of constructing a proof and thus knows that something is wrong. In practice, however, completeness bugs can be very annoying, difficult to detect, and time-consuming. A user may look for errors in the program to be verified or blame the annotation AUX when no proof is found for a correct program and try to improve AUX, while in fact nothing is wrong with it. It is therefore very important to systematically test for completeness bugs.

6.4. Axiomatization Coverage

We already motivated that testing the implementation part of a verification tool alone is not sufficient. Here, we would like to transfer some concepts of the usual software testing approach to our setting of testing a verification system. For the purpose of software testing, the VCG stage of a toolchain based verification systems like VCC can be seen as a compiler from a source language (here, C) to the target language (Boogie). An obvious starting point for testing strategies for the VCG part are regular compiler test suites. The quality of such a test suite in the regular software testing procedure is then assessed with the help of a code coverage metric.

However, code coverage is not an indicator for how well the declarative logical axioms and definitions are tested that define the semantics of programs and specifications and make up an important part of the system.

We give a definition of the notion of axiomatization coverage, measuring to which extent a test suite exercises the axioms used in a verification system. This declarative definition is based on the previous imperative version by Wagner [Wag09]. In addition, we make a clear distinction between completeness and soundness coverage in the following, as opposed to the single axiomatization coverage presented in his thesis.

The idea is to compute the percentage of axioms that are actually used in the proofs or proof attempts for the verification problems that make up a test suite. We distinguish two versions:

(a) the percentage of axioms needed to successfully verify correct programs (completeness coverage), and

(b) the percentage of axioms used in failed proof attempts for programs not satisfying their specification (soundness coverage).

An erroneous axiom may lead to unsoundness or incompleteness or both. The latter effect, where something incorrect can be derived and something correct cannot, is actually quite frequent. Because of that, completeness tests can reveal soundness bugs and vice versa. Nevertheless, one should use both kinds of tests (soundness and completeness) and, thus, both kinds of coverage.

For the remainder of this chapter, especially in the two case studies, we concentrate on the completeness version of axiomatization coverage. If not explicitly stated otherwise, "coverage" will stand for "completeness coverage" in the following.

6.4.1. Completeness Coverage

For the completeness version of axiomatization coverage, we define an axiom to be *needed* to verify a program, if it is an element of a minimal axiom subset, with which the verification system is able to find a proof. That is, if the axiom is removed from the subset, the verifier is not able anymore to prove the correctness of the program.

Definition 10. *A test case $P+(REQ \cup AUX)$ covers the axioms in a set Th if $Th \vdash P+(REQ \cup AUX)$ but $Th' \nvdash P+(REQ \cup AUX)$ for all $Th' \subsetneq Th$.*

According to this definition, not all axioms used in a successful proof for a test case are covered by that test. Some axioms may be redundant, i.e., they (a) can be replaced by (a combination of) other axioms used in the same proof, or (b) do not contribute to the proof because they were applied in "dead ends" of the proof search. For case (b), we argue that it is unlikely that the test case makes a relevant statement about the correctness of the axiom (other than that the axiom does not lead to inconsistencies in this proof). For case (a), we expect that there are different use cases in which to apply one or the other set of axioms and thus there should be test cases able to cover one specific set.

If we consider particular verification systems with a fixed axiomatization, we can define an axiom to be *strongly* covered, if it is needed in all proofs of the test case that the verification system is able to find using the given axiomatization.

Definition 11. *A test case $P+(REQ \cup AUX)$* strongly *covers the axioms in a set Th w.r.t. an axiomatization Ax, if $Th \subseteq Th'$ for all sets $Th' \subseteq Ax$ such that $P+(REQ \cup AUX)$ covers Th'.*

Our notion of completeness coverage is rather coarse in that it does not take the structure of the covered axioms (respectively the inference rules in case of the KeY system) into account. One obvious improvement would be to examine in the coverage analysis which part of an axiom is actually needed in a proof – e.g., in case of an implication $A_1 \vee \ldots \vee A_n \to B_1 \wedge \ldots \wedge B_m$, which A_k establishes the premiss and which of the B_k of the conclusion are actually needed further in the proof. Precise definitions of more fine-grained coverage metrics and their evaluation is part of future work.

Ideally, the coverage definitions would use logical entailment instead of inference. However, as we want to quantify axiom coverage in practice and verification tools are inherently incomplete, the coverage metric is based on the inference relation.

As a consequence, the axiom coverage of a test suite w.r.t. a system depends on resource constraints (e.g., number of proof steps allowed, timeout or memory limitations) and the implementation of the verification system, most notably the proof search strategy. This implies that when calculating axiom coverage, to get reproducible results, performance of the computer that the verification tool runs on has to be taken into account. In addition, axiom coverage of a test suite has to be recomputed not only when the axiomatization or test suite changes but also whenever parts of the implementation of the verification tool relevant for proof search are modified.

In general, the minimal set of axioms covered by a given verification problem is not unique. So, the question arises of what to do if a test case covers several axiom sets.

We chose to follow the more conservative approach to consider only one (non-deterministically chosen) axiom set to be covered by any given test case. A pragmatic reason for that choice is that it is very costly to compute all minimal axiom sets covered by a test case. But our conservative choice has another important advantage: If, for example, there is a logically redundant axiom A' that is an instance of some other axiom A, and there are verification problems that can be solved either using the more general axiom A or the more special A', then there are test cases that cover A and A' separately but not at the same time. We will count only one of A and A' to be covered by such a single test case. Now there are two situations to consider: (1) A' is included in the axiom set for good reason as it leads to better performance (shorter proofs) for a certain class of problems. Then there should be a test case in the test suite that can only be proved using A'. This test should have sufficiently low resource bounds so that it cannot be verified using A, and thus demonstrates the usefulness of A'. And there should be a different test case that does not fall in the special class to which A' applies and that can only be solved using A. Then, with the two test cases, both A and A' are covered. (2) A' is really redundant, i.e., it is not possible to construct a test case that can be verified with A' but not with A. Then A' is indeed akin to dead code (which also cannot be covered by test cases) and should be removed.

Orthogonal to the observations on which axioms are (strongly) covered by a test case, the reason *why* an axiom is covered by a test may be relevant. As we are interested in the interplay between the test program, its specification and the axiomatization of the verification tool, certain combinations of program and specification are unfit for testing the axiomatization. For example, consider the following trivial C implementation as a test case for the VCC system:

```
void test()
    _(ensures \forall int i; i != 0 ==> i / i == 1)
{}
```

This test case covers the identical axiom found in the axiomatization of VCC – but obviously such a test case is not particularly meaningful. While we can restrict test cases to exclude certain trivial combinations of requirement specification and program (e.g., no method contract should hold for the corresponding method body replaced by the empty statement block), we have to rely on the capabilities of the test author to provide sensible programs together with requirement specification.

Given such a pair $P{+}REQ$, we can then further restrict the set of auxiliary annotations (resulting in the test case $P{+}REQ \cup AUX$) to those that are essential according to

Definition 8 on page 83. Without this restriction, increasing completeness coverage is again trivial by adding the right auxiliary annotations that are not concerned with the correctness of the program and specification given.

6.4.2. Soundness Coverage

For the soundness version of axiomatization coverage, the above definition of *needed axioms* based on minimal axiom sets is not useful. The original definition of axiomatization coverage by Wagner circumvented the issue of soundness coverage as it did not distinguish explicitly between completeness and soundness – it was rather based on the idea that all axioms which change the output of the verification system if removed from the axiomatization are considered covered by a test case.

By monotonicity of logical entailment, for a sound verification system, removing axioms never alters the verdict of the verification system from the initial answer “not provable” to “provable”. Here, we assume that the verification system is able to give a definite answer for a soundness test case when using the complete axiomatization, as stipulated in Section 6.2. If the initial result of the verification tool is “timeout”, removing axioms may indeed change subsequent answers of the tool to “not provable”.

With this in mind, the only possible difference in prover output for a soundness test case is *which parts* of the annotations in the test case are reported by the tool as “not provable”. As noted by Wagner, for VCC the order in which the annotations are given matters and thus removing axioms may lead to formerly provable annotations appearing early in the test case to result in a counterexample. As a consequence, axioms considered to be covered by observing those changes in the prover output actually indicate *completeness* coverage for these formerly provable annotations.

Instead, we are only able to give the rather weak notion that an axiom is *used* in a failed proof attempt, if it occurs in the proof search, i.e., the verification system actively used the axiom for proof construction. What “used” means depends on the particular verification system and its calculus.

6.5. Case Studies

To assess the conclusiveness of the axiomatization coverage metric for completeness test cases, we conducted several case studies. In the first series of case studies by Wagner [Wag09], the axiomatization coverage of the VCC test suite tool was examined; additionally, external tests were used in order to reveal bugs in VCC. We extend on this exploratory work by testing the rule base of the KeY system – again, both using its own test suite and third-party tests. The results of the VCC case studies are also presented in the following for the sake of completeness and to provide a comparative value for the outcomes of the KeY case studies.

Computing Axiomatization Coverage in Practice. We have implemented a framework for the automated execution and evaluation of tests for both tools that computes the completeness version of axiomatization coverage, according to the following algorithm:

Algorithm 6.2 Algorithm to compute completeness axiomatization coverage

```
1: Inputs:
       Test case $T = P{+}SPEC$, set of axioms $Ax$
2: Output:
       Set of axioms covered by test case $T$
3: procedure COVERAGE($T, Ax$)
4:     $cov \leftarrow \varnothing$
5:     $(isProven, used) \leftarrow$ RUNTOOL($T, Ax$)
6:     if $\neg isProven$ then                    ▷ Wrong kind of test case or a bug in the tool
7:         return $\bot$
8:     end if
9:     $rem \leftarrow used$
10:    for all $t \in used$ do
11:        $rem \leftarrow rem \setminus \{t\}$
12:        $(isProven, \_) \leftarrow$ RUNTOOL($T, cov \cup rem$)
13:        if $\neg isProven$ then
14:            $cov \leftarrow cov \cup \{t\}$
15:        end if
16:    end for
17:    return $cov$
18: end procedure
```

To compute an approximation of the axiomatization coverage for a completeness test case $P{+}SPEC$, the procedure is as follows: in a first step, $P{+}SPEC$ is verified with the verification tool using the complete axiom base available (line 5 in Algorithm 6.2). Besides gathering information on resource consumption of this proof attempt (e.g., number of proof steps respectively time needed), information on which axioms are

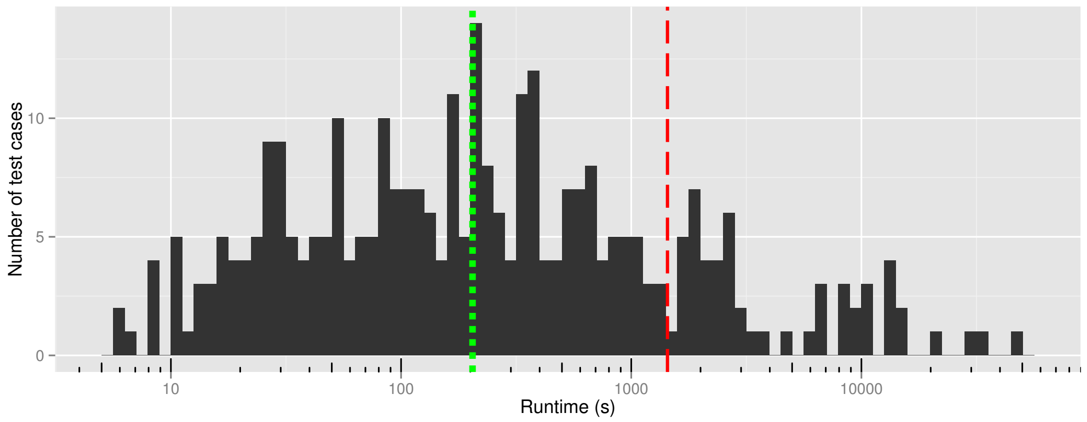


Figure 6.3.: Frequencies of runtimes for test cases of the KeY test suite. Note the logarithmic scale of the x-axis. The dotted green line indicates the median of the runtime set (at 204s), the dashed red line corresponds to the mean (1437s).

actually used in the proof are recorded as set called $used$ (e.g., by leveraging Z3's option to generate unsatisfiability cores in case of VCC,[1] respectively parsing KeY's explicit proof object). This set $used$ is a first approximation of the completeness coverage of the test case but has to be narrowed down in a subsequent step to yield the actual axiomatization coverage.

For this, in a reduction step (beginning with line 9), we start from the empty set cov of covered axioms and the set $rem = used$ of axioms remaining to be checked for coverage. For each axiom t in the set of axioms $used$ from the first proof run, an attempt to prove $P{+}SPEC$ using axioms $cov \cup (rem \setminus \{t\})$ is made. If the proof does not succeed, t is added to set cov. Axiom t is removed from rem and the next proof iteration starts until $rem = \emptyset$.

In all these subsequent proof runs, resource constraints are set to twice the amount of resources needed for the first proof run recorded initially. This allows us to calculate axiom coverage in reasonable time and ensures comparability of coverage measures between computers of different processing power.

The resulting set of axioms cov is only an approximation of the coverage of $P{+}SPEC$. For precise results the above procedure would have to be repeated with cov as input until a fixpoint is reached. For practical reasons we compute axiomatization coverage using only one iteration for the following case studies.

In the VCC case study performed by Wagner [Wag09], even without the performance improvements introduced later by Beckert et al. [BBW13a] as described above, computing a minimal axiom set in average only takes several minutes. For the case studies involving the KeY system, the average test case has similar runtime compared to the VCC tests. However, the distribution of runtimes, given in Figure 6.3, shows that a number of tests run for more than an hour and some outliers are even beyond this mark with a maximum runtime of 47890 seconds.

[1]The coverage experiments in this chapter have been produced by an older version of our framework without this feature – however, this only impacts performance of the framework.

The overall runtime for the complete test suites is still acceptable as the coverage computation can be parallelized (see Section 6.5.2) at the granularity of single test cases. In our case, total runtime is thus determined by the single test case with the largest runtime. Possible speed-ups, e.g., by using data from previous test runs or improved parallelization are discussed further in Section 6.6.

6.5.1. Testing the Axiomatization of VCC

Using the testing framework presented above, a first set of experiments was performed by Wagner [Wag09] for the VCC tool, examining the completeness axiomatization coverage. For comparison, a second set of experiments was conducted [BW10], measuring both axiomatization completeness coverage and code coverage of an updated VCC version. We only give a brief summary of the results in the following as a reference value for the later experiments conducted for the KeY system – for a more detailed analysis, we refer to the thesis of Wagner, as well as our previously published work [BW10; BBW13a].

First Experiment. The first experiment by Wagner consisted of computing completeness coverage for two versions of the VCC tool with corresponding axiomatization. For both coverage computations, the same test suite from one of the VCC version was used. This test suite, taken from VCC version 2.1.20731.0 contains 202 completeness tests (as well as other tests not applicable to coverage computation).

Completeness coverage for VCC version 2.1.20731.0, as well as for version 2.1.20908.0 (about six weeks further in development) was computed – with results as given in Table 6.4. The decreased coverage from the earlier VCC version to the more current one was determined to result from modifications in the axiomatization.

To ascertain whether axiomatization coverage is a valid quality metric for test suites, a correlation between the coverage and the number of bugs left in the axiomatization would have to be established. A first indication for such a correlation are completeness test cases that indeed uncover a soundness or completeness issue of the verification tool. However, the existing test suite of the VCC system was unsuitable for this purpose – as the tests are run as part of the VCC build process, any issue identified with the help of the test cases is no longer present in the published VCC distribution. Instead, additional test cases are needed to uncover bugs in the axiomatization – with the help of third party test cases, Wagner was able to identify one completeness issue in the VCC tool chain.

Second Experiment. In a second experiment, we used a more recent implementation of VCC (version 2.1.30820.1) published one year after the VCC version used in the first experiment, together with the accompanying, updated test suite. The examined part of the new test suite that corresponds to the test suite used in the first experiment had been updated and now contains a total of 698 test cases, 417 of which are completeness tests (the rest are soundness tests and tests checking for parser errors). The axiomatization now consists of 439 axioms. Of these, 211 were covered by the completeness tests (48%), i.e., axiomatization coverage has increased again to the level of VCC version 2.1.20731.0, due to the updated test suite.

Earlier version of axiomatization

	Total	Covered	Percentage
axioms for C language features	212	84	40%
axioms for specification language features	166	102	61%
all axioms	378	186	49%

Later version of axiomatization

all axioms	384	139	36%

Table 6.4.: Axiomatization coverage measures for the first VCC experiment

Further, this new version of the test suite contains an additional directory of test cases we did not consider here to be able to compare coverage results with the first experiment. We thus expect even better coverage when taking all tests into consideration.

For this second experiment, we additionally computed the code coverage[2] for the part of VCC that is related to the semantics of the programming and the specification language, i.e., the verification condition generator. Using the same test suite of 417 completeness tests, the resulting code coverage turned out to be 70%. This is an interesting result as it shows that axiomatization coverage (48%) can be quite a bit lower than code coverage (70%). And it is evidence that axiomatization coverage is independent of code coverage. Therefore, axiomatization coverage should indeed be considered in addition to code coverage to judge the quality of a test suite.

Additional investigations showed a further difference between code coverage and axiomatization coverage: The code coverage of *individual* test cases is higher than their axiomatization coverage. Axiomatization coverage can be as low as 1% for some tests, while code coverage is never less than 25%. That is, there is a certain amount of "core code" exercised by all tests, while there are no "core axioms" used by all tests (this may, of course, be different for other verification systems).

6.5.2. Testing the Calculus Rules of KeY

The KeY System

Proving validity of proof obligations with the KeY system is done using automatic proof strategies which apply sequent calculus rules implemented as so-called *taclets*. The set of taclets provided with KeY plays a similar role as the prelude in case of VCC, as it captures the semantics of Java, built-in abstract data types like sequences etc. In comparison to the prelude of VCC, however, KeY also contains taclets that deal with first order logic formulas, whereas first-order reasoning in VCC is handled by the SMT component Z3.

[2]Code coverage was computed using the code coverage feature available in Microsoft Visual Studio 2010 Premium.

The development version of KeY as of August 2012 contains about 1500 taclets. However, not all of them are available at a time when performing a proof, as some taclets exist in several versions, depending on proof options chosen (e.g., handling integer arithmetic depends on whether integer overflows are to be checked or not).

Automatic proof search is combined with interactive steps of the user, in case a proof is not found automatically. For our purposes, the interactive part of KeY is irrelevant, as we restrict test cases to those that can be proven automatically – otherwise, finding a minimal set of taclets needed to prove a program correct is infeasible.

Computing Axiomatization Coverage with KeY

The procedure to compute taclet coverage for test cases with KeY used here is similar to reducing VCC's axiomatization – but whereas the implementation for VCC makes use of Z3's unsatisfiability core, here, the initial set of taclets to start from is directly taken from the explicit proof object that KeY maintains and that is used to save and load (partial) proofs. This object stores sufficient information about each taclet application in the sequent calculus proof, such that the proof can be reconstructed by KeY without performing proof search. These taclet applications recorded normally contain a lot more taclets than are actually relevant for finding the proof. Thus, we reduce the set of taclets used in the proof construction one by one in a second step in the same manner as for VCC.

In the case of VCC, a reduction step is performed by removing one axiom from the file storing the axiomatization and applying VCC to the test case (using the modified axiomatization), with the advantage to not having to modify VCC to be able to compute axiom coverage. This is only viable if invoking the verification tool is reasonably fast, as is the case with VCC (which was designed with the goal of fast response times in mind, deemed necessary for auto-active verification).

The time to start up KeY, while insignificant in interactive verification, prohibits employing the same reduction approach as used for VCC. Most of these start-up costs are incurred by loading the set of taclets, which is only performed once – afterwards, each proof obligation loaded in an interactive session makes use of the pre-loaded taclets. In order to account for these performance characteristics, we modified KeY to use only a subset from the set of all loaded taclets by the strategy in proof search. This subset can be defined on a per proof basis and allows performing all reduction steps of a test case within a single KeY instance.

Even with this improved setup, computing the taclet coverage of the test cases supplied with the KeY package is too time consuming on a single computer. Therefore, we extended our taclet coverage computation framework to make use of cloud computing by distributing tasks on multiple computing resources (via Amazon's Elastic Compute Cloud (EC2), together with other infrastructure provided by the Amazon Web Services).

The smallest task distributable to a computing unit in our implementation is to compute taclet coverage of a single test case. Task descriptions simply consist of the file name for which coverage is to be computed. All tasks are stored as messages in a global queue that can be accessed concurrently by multiple clients; this functionality is provided by the Amazon Simple Queue Service (SQS). Progress for each single reduction step of a client is recorded in a single, global repository (using the Amazon Simple Storage Service). Besides some statistical data, all information necessary to resume an aborted reduction run is stored.

Obviously, with enough computing instances, "wall clock" execution time of the overall coverage computation is mainly determined by the execution time of the most expensive proof task. Indeed, for the test cases delivered with KeY, few such complex test cases are the bottleneck in the reduction process. In order to improve on this, one option is to split such tasks into more fine granular subtasks. Instead of each computing unit sequentially working on reducing one taclet at a time for a given test case, we could divide parts of the task as follows: while one instance (called A) still sequentially reduces taclets for the test case, another set of instances may try to eliminate a single taclet using *all but one* taclet from the last known working taclet configuration. If KeY cannot close the proof without this taclet, we know that the taclet is covered by the test case and we can communicate this fact to instance A; otherwise, instance A still has to check whether the taclet is necessary for the proof with its *current* set of taclets used.

Source code of the KeY version used for computing taclet coverage can be found at `http://formal.iti.kit.edu/~bormer/coverage/`, together with scripts necessary to run coverage computation on the Amazon cloud infrastructure and an application that provides a web-interface to inspect the current progress of a running reduction process.

Axiomatization Coverage Results

Using the version of our testing framework presented in the previous section, we automatically executed the test cases contained in KeY's test suite, as well as parts of a custom Java compiler test suite and measured the taclet completeness coverage.

Third Experiment. In this experiment we used the development version of the KeY tool[3] as of August 16, 2012. As part of the KeY source distribution, a test suite is provided containing 335 test cases of which 327 are completeness and 8 are soundness tests. The complexity of the proof obligations ranges from simple arithmetic problems to small Java programs testing single features of Java, up to more complex programs and properties taken from recent software verification competitions.

From the 327 completeness tests of the KeY test suite, we computed the taclet completeness coverage of 319 test cases testing verification of functional properties – another eight test cases are concerned with the verification of information-flow properties and were omitted due to resource constraints.

[3] Available at `http://i12www.ira.uka.de/~bubel/nightly/`

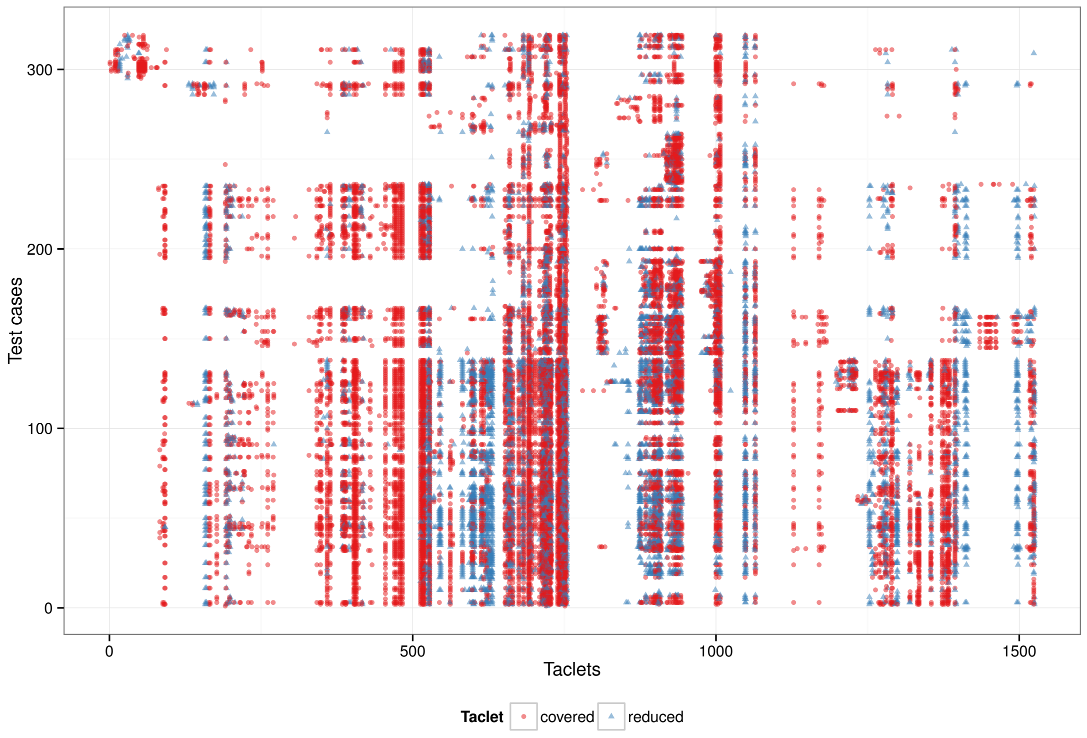


Figure 6.5.: Coverage data for the KeY test suite. Red dots indicate a taclet covered by a test case, blue dots are taclets used in the initial proof for the test case but reduced in coverage computation.

The test runs were distributed on multiple computers using Amazon's Elastic Compute Cloud service, taking approx. 135 EC2 Compute Unit[4] hours to complete in total.

The overall taclet coverage we computed for the 319 completeness tests was 38% (with 585 out of 1527 taclets covered). The raw data obtained by coverage computation is depicted in Figure 6.5 – of course, this presentation form only allows a first estimation of coverage distribution along the different taclets. In addition, the diagram gives an idea of the effectiveness of using the proof object of the initial proof as a starting point for taclet reduction: All taclets eliminated in the coverage computation are shown in blue, while the actually covered taclets correspond to the red dots.

At a more aggregated level, Figure 6.7 shows a histogram of the number of test cases each taclet is covered by. The overall coverage seems to be comparable to the coverage results gained from the VCC test suite. However, in contrast to VCC, KeY also performs first-order reasoning with the help of taclets instead of using an SMT solver – for better comparison with VCC, all such taclets would have to be excluded from coverage computation.

[4]According to Amazon's EC2 documentation, "One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor."

While some features of Java or JML are not covered at all by the current test suite of KeY, for other taclets, low coverage might result from:

(a) the fact that some taclets have been introduced just recently, for a particular use case (e.g., taclets enabling automatic induction proofs in certain cases) and no test cases have been written yet,

(b) redundant taclets used to shorten the proof (e.g., the KeY taclet replacing "$F \rightarrow \mathrm{true}$" by "$\mathrm{true}$" for any FOL formula F is made redundant by the taclet replacing implication by its definition, together with the taclet rewriting the result "$\neg F \vee \mathrm{true}$" to "$\mathrm{true}$". This is one example for a taclet used but *not* covered in a proof.) and

(c) obsolete taclets that are still contained in the rule base.

Measures to handle cases (b) and (c) have already been discussed in Section 6.4.1. For case (a), a review and testing process has to ensure that the axiomatization coverage for newly introduced taclets is increased by writing specific test cases.

In order to perform a more detailed analysis, coverage results were examined by groups of taclets respectively test cases. For grouping taclets, we used the already existing structure given by the organization of taclets in different files in the KeY distribution – similar taclets (e.g., taclets handling Java language features or taclets for propositional logic) are contained together in one file.

Histograms of the number of test cases a taclet is covered by, split by taclet group, already allowed us to compare the quality of the test suite w.r.t. the different groups: not surprisingly, taclets handling propositional logic or Java heap properties are covered quite often (in both groups over 60% of taclets are covered at least in one test case). On the other end of the spectrum, we were able to locate underrepresented taclet groups, e.g., relevant for Java assertions or the `bigint` primitive type of JML (with coverage of each group below 10%). This coarse classification already allows us to focus the effort of writing new test cases on constructing specific tests for rarely covered taclet groups.

In order to group similar test cases together to identify commonalities, we used the R environment for statistical computing [RCT12] to cluster test cases. The result of this clustering is shown in Figure 6.6.

Two of these test case clusters are notably representative for different types of test cases: while the tests in Cluster 5 mostly encompass a small set of related taclets, tests in Cluster 3 span almost the entire taclet base of KeY. Indeed, test cases belonging to Cluster 5 have been written as specific tests for single features of KeY – in this case, integer arithmetic and handling strings (which need similar taclets because the functions retrieving characters from a string or getting substrings use integers; the small group of taclets in the upper left corner of Cluster 5 are the taclets handling strings). Test cases corresponding to Cluster 3 are mostly taken from verification competitions dealing with data structures on the heap like linked lists.

We believe that a good test suite for a verification system needs both of these types of test cases *for each taclet*. The need for broader test cases, covering several combinations of taclets, is supported by studies (e.g., by Kuhn et al. [KWG04]) which show that software

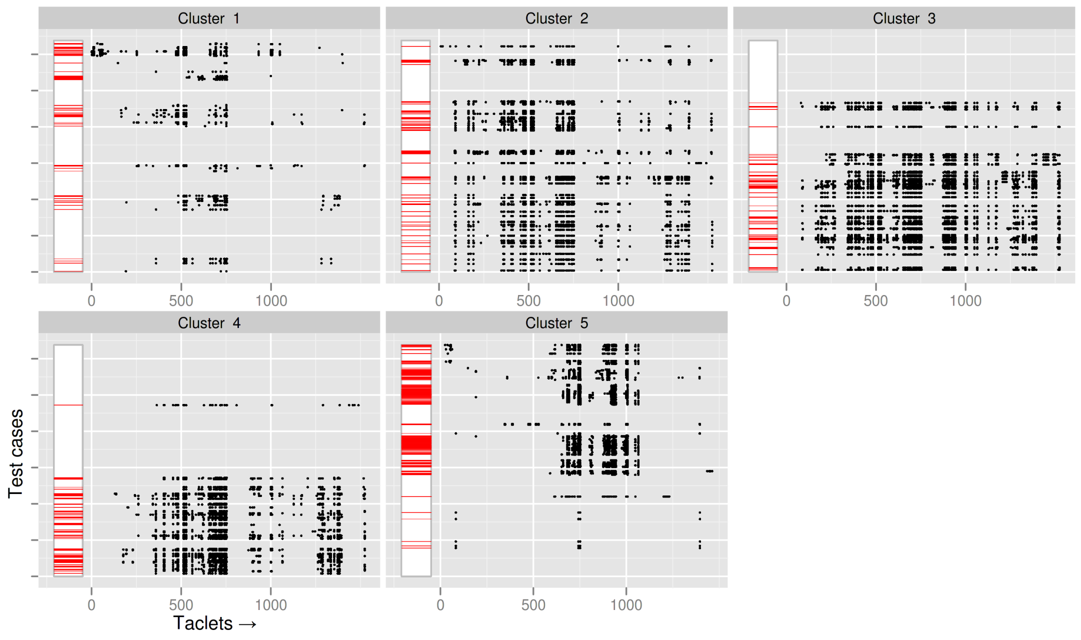


Figure 6.6.: Groups of similar KeY test cases in terms of taclet coverage. Each point in the diagram indicates a taclet covered by a test case. Clusters are computed by hierarchical clustering using Ward's method and Jaccard distance as measure for dissimilarity between taclet coverage of two test cases. The x-axis shows taclets sorted by group, the y-axis shows test cases sorted by directory. If a test case is contained in a cluster, this is indicated by a red line in the box on the left.

failures in a variety of domains are often caused by combinations of several conditions. Specialized test cases, in comparison, might simplify testing different aspects of one taclet by being able to better control the context in the proof a taclet will presumably be applied in. As a measure for this, we define the *selectivity* of a test case as the number of taclets covered by the test.

The current state of the KeY test suite w.r.t. this selectivity criterion is shown in Figure 6.8. For each taclet, the average selectivity of all test cases covering this taclet is shown, together with the population standard deviation from the average. The leftmost taclets in this diagram are good candidates for which additional test cases might be needed, as they are only covered by specialized test cases. Also, taclets with a high selectivity average of the corresponding test cases but low deviation indicate need for improvements, as only broad test cases cover the taclets.

Fourth Experiment. The last experiment we conducted used parts of the test suite of the Java2Jinja[5] compiler [Loc12] in order to increase taclet coverage compared to KeY's own test suite.

[5]See http://pp.info.uni-karlsruhe.de/projects/quis-custodiet/Java2Jinja

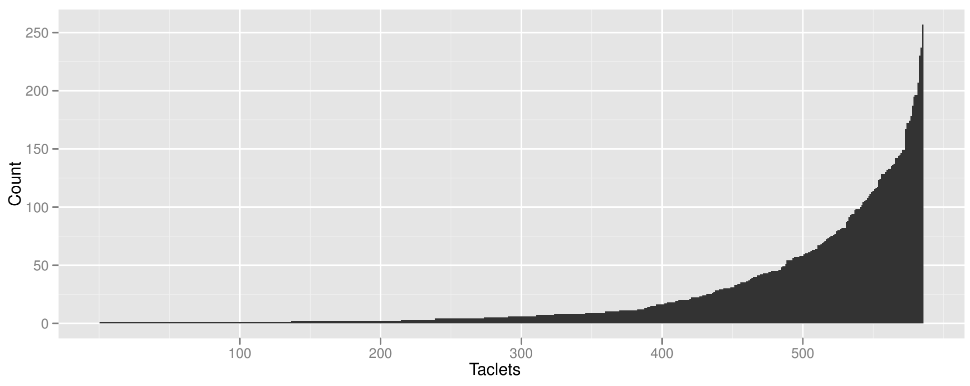


Figure 6.7.: KeY taclet coverage counts (y-axis shows number of test cases a taclet is covered by; x-axis shows taclets at least covered by one test case, sorted by y values).

The part of Java2Jinja's test suite we considered is hand-written and consists mostly of small Java programs testing few Java features at a time, often dealing with corner cases of the Java semantics. All tests are run by providing concrete, fixed input parameters to the test's main method. The result of the execution of a test is an output to the console – the expected outcome of the test case is given as annotation in the test file. This annotation can easily be converted into a postcondition suitable to be proven with KeY.

From a total of 43 annotated test cases, 21 were applicable to KeY – most of the other tests included features not yet supported by KeY (e.g., Java generics). Of the 21 suitable tests, 12 were directly provable (two thereof using user interaction), the rest of the proofs exceeded allocated time or memory resources.

One result of the test runs was that the Java2Jinja test suite covered 195 of 1527 taclets, corresponding to 13% taclet coverage. Even this small set of additional test cases covered nine taclets that were not covered by KeY's own test suite.

Errors Found in KeY

As already argued in Section 6.5.1, we could not expect to find soundness bugs using KeY's own test suite. Therefore, we performed the last experiment using Java2Jinja's test cases – resulting in two bugs found in the rule base of KeY.

The first bug we discovered is related to implicit conversion of integer literals to strings in Java: In case a negative integer literal is converted to its string representation, the minus sign is placed at the last instead of first position in the resulting string. The Java2Jinja test case that revealed this bug has not been written to test exactly this conversion feature but rather the correct handling of precedences in integer arithmetic. To write an additional test case for this exact feature that increases axiomatization coverage is easy, as exactly one taclet is involved in the bug and the condition under which this taclet is applicable is clearly visible to the user.

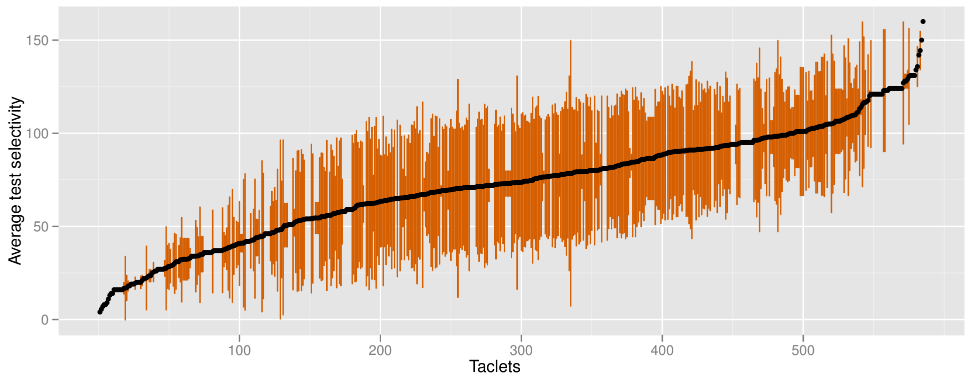


Figure 6.8.: Average test case selectivity by taclet. Black points: average selectivity (see Sec. 6.5.2) of all test cases covering a taclet. Population standard deviation of this value from the average is shown as red bars.

The second bug found in the taclets of KeY deals with the creation of inner member classes, using a qualified class instance creation expression. An example of a qualified class instance creation can be seen in Listing 6.9, a modified version of the Java2Jinja test case that revealed a bug in KeY. In this example, the instance creation expression is `o.new Inner()` and `o` is the so-called qualifying expression. The expected result of this test case is that the instance creation expression throws a `NullPointerException`, as `o` is null, in accordance to the Java language specification (Sec. 15.9.4):

> [. . .] If the qualifying expression evaluates to null, a NullPointerException is raised, and the class instance creation expression completes abruptly.

However, the corresponding taclet in KeY that symbolically executes this instance creation expression does not check whether the qualifying expression is null and always creates an object of the inner class without throwing the required `NullPointerException`.

While both of these bugs in the taclets allow the user to prove properties about certain programs that do not hold, they do not lead to unsoundness in the general case: both bugs are only triggered if the Java program contains the corresponding features (i.e., qualified class instance creation or conversion of negative integer literals to strings). In addition, the bugs only influence the correctness of the proof if the property to be verified relies on those features.

In both cases, the KeY test suite did not cover the relevant taclets at all, and the increase in the axiomatization coverage by the two relevant Java2Jinja tests indeed allowed us to reveal faulty taclets. This shows that axiomatization coverage is a useful metric to get a first hint to parts of the axiomatization that may be target for further inspection.

```java
public class InnerClassOuterNull {
  public static void main(String[] args) {
    InnerClassOuterNull o = null;
    try {
      o.new Inner();
    } catch (NullPointerException e) {
    Sys.out.println("NPE thrown");
    }
  }

  class Inner { Inner() {} }
}
```

Listing 6.9: A test case for qualified class instance creation (modified version of a Java2Jinja test case).

6.6. Improving Performance of Axiomatization Coverage Computation

The current approach for computing the axiomatization coverage, even with the improvements made compared to the method by Wagner [Wag09], is rather time-consuming. While being fast enough to be used for daily regression testing of the axiomatization, the runtime of the coverage reduction renders more elaborate experiments impractical. As explained in Section 6.5, computing the minimal set of covered axioms of a test case requires reducing the set of needed taclets until a fixpoint is reached, but this computation was omitted in favor of receiving results in a timely fashion. Another example is the generation of new regression tests for the KeY tool, as presented in previous work [BBW13b], despite using around 200 cores of Intel XEON E5430 processors at $2.66\,\mathrm{GHz}$ in a cluster of machines, runtime of the coverage computation was cut off at $24\,\mathrm{h}$.

To improve on the time needed for axiomatization coverage analysis one option is to use heuristics for the (set of) axioms to remove from the axiomatization in the next reduction step. In previous work [BBW13b], we examined three different heuristics to generate regression tests: (a) removing single axioms in lexicographical order, as a "naive" approach for comparison, (b) removal of single, random axioms and (c) removal of up to six random axioms in one reduction step, together with "backtracking" by adding back half of the axioms removes in case the proof does not succeed.

Besides these straightforward heuristics, other options include to incorporate coverage results from earlier completed coverage computations in the heuristic. For identical test cases in two coverage computation instances, one obvious optimization is to start reduction with the set of covered axioms of the first computation. Also, for new test cases, coverage data of previous computations could be of benefit – e.g., using association rule learning, one could identify whole groups of axioms to be removed in one step.

Divide and conquer algorithms, e.g., akin to binary search, seem to be suited to reduce computation times at first glance. However, they do not help in practice: as the reduction step does not start from the whole axiomatization but rather from the subset T of axioms actually used in a proof, only relatively few axioms remain that are *not* covered and can be discarded in the iterative proof runs. For divide and conquer algorithms to be successful, large sets of axioms which could be discarded at once are needed.

While divide and conquer does not work, it is possible to use additional CPU resources in exchange for better "wall-clock" time by parallelizing the task on a more fine-grained level than single test cases. The idea is to use (at least) two additional execution threads for a test case: while one thread (T_1) tries to prove the test case with the axiom set $Axs = \{Ax_2, \ldots, Ax_n\}$ (i.e., testing whether Ax_1 is covered), the two other threads at the same time use sets $Axs \setminus \{Ax_2\}$ and $(Axs \setminus \{Ax_2\}) \cup \{Ax_1\}$ as starting point for the coverage computation step. Depending on the outcome of thread T_1, the corresponding result of one of the other thread is dismissed.

Note that the technique presupposes a fixed order in which the axioms are reduced. Also, the method is worthwhile only for test cases with large runtimes of each single reduction step – whether a test case would benefit from further parallelization can be determined after the first few reduction iterations.

6.7. Improving Completeness Coverage of Existing Test Suites

We mentioned in Section 6.3 that the author of a test case may expect the verification tool to conduct a proof in a certain manner, e.g., using a specific set of axioms. By computing the completeness coverage of a test case, we provide such an axiom set that can be inspected in order to detect discrepancies between actual result and the expectations. In addition, the set of covered axioms can be used to form a more precise regression test, defined as follows:

Definition 12 (Completeness regression test case)**.** *A completeness regression test case for a verification system is a tuple* $< T, Th >$ *where* $T = P{+}SPEC$ *and* T *covers the axioms in* Th*.*

If the set of covered axioms Th in the previous definition is weakened to a superset of Th, we call the corresponding test case a *weak completeness regression test*. Weak completeness tests take into account that the prover may non-deterministically choose alternative set of axioms in a proof without signifying a completeness issue of the verification system.

As stated in Section 6.4.1, the set Th of axioms covered by a test case is, in general, not unique. Therefore, a single regular completeness test case can be extended to multiple regression tests, given the different sets of covered axioms. To compute some of these coverage sets, we presented in previous work [BBW13b] the idea of reusing results of coverage computation, as shown in Algorithm 6.10: starting with test case T and the set Th of covered axioms as a result of Algorithm 6.2 on page 97, we repeat coverage computation with the same test case T but using the complete axiomatization minus one of the elements of Th. As the set Th is minimal, if the proof of T with the reduced axiomatization succeeds, we obtain a new set of axioms Th' covered by test T. This process can be repeated with all axioms from Th, and recursively with the newly found coverage sets.

Algorithm 6.10 Algorithm to generate completeness regression tests

```
 1: Inputs:
        Test case $T = P{+}SPEC$, set of axioms $Ax$, initial set of axioms $Th$
        covered by $T$
 2: Output:
        Set of sets of axioms covered by test case $T$
 3: procedure RegressionCoverage($T, Ax, Th$)
 4:     $res \leftarrow \{T\}$
 5:     $next \leftarrow$ choose $x \in Th$
 6:     $cov \leftarrow$ Coverage($T, Ax \setminus \{next\}$)
 7:     if $cov \neq \bot$ then
 8:         $res \leftarrow res \cup \{cov\}$
 9:     end if
10:     return $res$
11: end procedure
```

In practice, the reduction in line 6 is still time consuming and thus only some coverage sets can be determined – choosing a good axiom candidate to remove from the axiomatization in line 5 is thus crucial in order to get usable results. For this, we have examined three heuristics for selecting axioms from Th in the reduction step in [BBW13b] w.r.t. the completeness coverage of the union of all regression test cases found for a given test case. As a result of our experiments, with the union of all results produced by the different heuristics we were able to increase coverage of the KeY axiomatization from 585 taclets (see Sec. 6.5.2) to 721 taclets using regression test cases.

6.8. Related Work

In principle, instead of or in addition to testing, parts of verification tools (in particular the axiomatization and the calculus) can be formally verified. For example, the Bali project [Ohe01], the LOOP project [JP04], and the Mobius project [Bar+06], all aimed at the development of fully verified verification systems. Calculus rules of the KeY verification system for Java were verified using the Maude tool [ARS05].

Concerning VCC, Böhme et al. report in 2008 on the verification of an earlier version of the VCC prelude (containing "about 450 axioms" [Böh12]) using their framework HOL-Boogie [BLW08]. The HOL-Boogie framework allows reasoning about Boogie programs in Isabelle/HOL – for this, a custom prover back-end for the Boogie tool exports the verification conditions for the program as generated by Boogie which can then be imported into Isabelle/HOL with the help of a loader. As the VCC prelude is also given in form of a Boogie program, this allows conducting proofs about the background axiomatization of VCC in Isabelle/HOL with the help of this framework.

The approach used by Böhme et al. to prove consistency of the VCC axiomatization is to manually construct a consistent theory in Isabelle/HOL by conservative extensions, defining types, operations etc. as used in the VCC prelude. The axioms of the prelude, imported as proof obligations into Isabelle/HOL, are then derived in this theory. Regarding the verification effort, the authors intended to conduct this consistency proof for a newer version of the background axiomatization as part of future work with an estimate of “several man-months” [BLW08] to complete the proofs.

Verifying a verification system is useful, but it cannot fully replace testing; this is further discussed by Beckert and Klebanov [BK06]. The authors claim that verifying a verification tool involves a huge effort that is, to some extent, better directed to improve other qualities of verification systems relevant in practice (e.g., efficiency of the tool). Also some sort of cross-validation between tools is needed, amongst other reasons, as there is no single, authoritative formal language specification for Java. For this, one option mentioned explicitly by Beckert and Klebanov [BK06] is to use “cross-validation with test programs written by different people.”

For the theorem provers and SMT solvers that are components of verification systems, there are established problem libraries that can be used as test suites, such as the SMT-LIB library [Bar+]. Alternatively, the results of SMT solvers can be validated using proof checkers. For example, Z3 proofs can be checked using Isabelle [Böh09]. Another example is the Formally Verified Proof Checker that was implemented in ML and formally verified using HOL88 [Wri94].

An interesting application of conformance testing is the official validation test suite for FIPS C (a dialect of C) [Jon97]. To determine how well this test suite covers all features of the C language, a reference implementation of a C compiler was built such that the implementation modules of the compiler could be associated to parts of the C standard. This allowed relating code coverage of the reference compiler to coverage of the language standard when compiling the programs of the test suite.

In the case studies presented in this chapter, the test cases used for coverage analysis were written by hand – coming up with meaningful test cases in this way is a time consuming process and complementing alternatives are worthwhile.

One option to obtain test cases is to use randomly generated programs with known behavior as a basis for comparing the outcome of the tool under test with the expected behavior of the generated program [Cuo+12b]. In order to test parts of Frama-C, the tool Csmith [Yan+11] was used to generate random C programs. These C programs output a checksum of their global variables, allowing to compare execution of the program compiled with a reference compiler to analysis results obtained with Frama-C.

Another approach to identify erroneous axioms is to use model-based testing [AD10]. In order to test a first-order logic axiom, the user provides an interpretation of the functions and predicates used in the axiom by giving for each a Haskell implementation. The axiom to be tested is then translated into an executable Haskell function (using the provided interpretation of functions and predicates). This function is then tested to be true for a set of generated test inputs with the help of a standard Haskell testing framework.

6.9. Conclusions and Future Work

Improving the quality of verification systems is an important problem in practice. Based on previous work, we have presented a refined definition of the axiomatization coverage metric for testing verification tools. We claim that this metric allows assessing and improving on the quality of test cases for a range of deductive verification tools.

This claim is supported by a first set of smaller case studies, extending on previous experiments by Wagner [Wag09] both for the VCC tool and KeY system. As a result, all experiments showed that the axiomatizations of the systems under test were exercised to a rather low degree, as indicated by the overall axiomatization coverage for the examined test suites. A more detailed analysis was performed for the KeY system and corresponding test suite – among others, different types of test cases w.r.t. coverage were identified and the notion of test selectivity was introduced. To evaluate the usefulness of the completeness version of axiomatization coverage, in a first step, we tried to increase coverage by additional test cases to uncover erroneous axioms. For this, we used parts of the Java2Jinja test suite to test the KeY tool, which revealed two bugs in KeY's rule base.

By enriching the test results of a test case for deductive verification systems with the expected set of axioms used in a proof, we were able to generate additional regression test cases from the existing test suites.

To facilitate further experiments, we improved on the runtime of the coverage computation by using information from the proof process in the axiomatization reduction step. Further ideas to speed up this computation not yet implemented were also discussed.

Already this coarse coverage criterion can be used as a first measure to judge and improve the quality of existing test suites. Further coverage statistics, like the test case selectivity, may be used additionally to identify axioms that are underrepresented in the test suite. Also, clustering test cases similar in their axiomatization coverage may hint at which kind of additional test cases are still missing (e.g., large and complex programs using many program language and specification language features at once; or rather small, specific test cases covering few and similar axioms).

Here, we only provided a preliminary examination to find out whether axiomatization coverage is a worthwhile metric to be analyzed further. It seems plausible to ensure that there is a test case for each axiom; also, we have seen that current test construction methods for tools in practice do not result in test cases with high coverage, making the need for further test cases evident. However, we still don't know whether writing test cases specifically to increase coverage is better in finding issues in the axiomatization than constructing an *arbitrary* additional test case. Further case studies are needed to demonstrate differences between different test construction methods with statistical significance. The same holds for examining the relevance of our second proposed metric, the test selectivity. Also, the rather coarse notion of completeness coverage presented is unlikely to be sufficient to evaluate the benefit of specifically testing certain axioms – a more fine-grained axiomatization coverage criterion has to be developed and examined with our testing framework. To concentrate on relevant and actually used taclets in these case studies, we plan to investigate the reasons why some axioms are not covered, e.g., with the help of developers of the verification systems.

Another shortcoming is the current definition of soundness coverage which is rather weak and should be improved. For this reason, also the case studies conducted so far do not yet include measurements for soundness coverage and examination would have to be extended in future work. Further, a more detailed investigation on how often completeness bugs also manifest themselves as soundness issues is in order.

One recurring issue in testing verification systems is to obtain meaningful test cases. For a completeness test case, three components are needed: the program, together with a requirement specifications and a set of auxiliary annotations, as well as a test oracle that is used to judge the expected test outcome. While there is an abundance of programs suited as a basis for test cases, the other ingredients are hard to come by. Most of the tests used in our case studies were written by hand specifically for the verification system in question. Transferring these tests to be applicable for other verification system is non-trivial, e.g., due to differences in the specification language. To test axioms relevant for the semantics of programming language features (respectively rules for symbolic execution of Java programs in KeY), one approach would be to automate the manual construction of test cases as shown at the Java2Jinja compiler test suite. A simple requirement specification could then be obtained by existing test generation tools, providing the input to the test program – running the program with the input then provides the expected output respectively postcondition.

7. Early and Precise Feedback for Deductive Verification

Contents

One of the issues in specifying and verifying large software systems are the complex interdependencies of different annotations – a problem we call "entangled specifications" and which was briefly discussed in Chapter 4. Useful feedback of the deductive verification tools is, by their modular design philosophy, often only possible after the user has annotated all program parts the current module depends on. Even after providing sufficiently many annotations, another consequence of modular verification is that mismatching specifications are not directly apparent (e.g., the specification of a module may fit at one call site, but may not be strong enough to establish properties in other program contexts). Both issues may lead to several iterations of changes to all involved specifications until a fix-point is reached that allows verification of the full system.

Here, we will introduce a technique that is able to provide the user with feedback as soon as an annotation has been completed, helping to avoid misconceptions about the behavior of the program, respectively specification. Errors spotted and corrected in this early phase of annotation development help to avoid writing mismatched specifications and thus reduce the number of iterations needed to adapt annotations to each other.

Our method makes use of software bounded model checking (SBMC) [CKL04], which is able to analyze a system beyond module boundaries, by using techniques such as inlining method calls – these procedures are not available for deductive verification tools due to scalability issues. This ability, however, does not come for free: The specification languages of SBMC tools are not as expressive, and they have less precision as compared to deductive verification systems (because of the bound imposed on the length of the program execution trace).

In the following we explore a combination of deductive verification and SBMC to obtain an improved verification process, where SBMC is used to support the user in finding module specifications suitable for the different contexts the module is used in, while deductive verification provides the final correctness proof. For this, auxiliary specifications that the user adds as annotations to the system are translated into input for the SBMC tool. As the user's annotations are aimed at deductive verification, the SBMC tool may not be able to handle them, but in many cases it can provide early feedback. In particular, SBMC can check the appropriateness of the specifications beyond the boundaries of a single module – even if other relevant modules are not specified (yet). This allows testing early on whether the different module specifications in the system match the implementation at every step of the specification process.

As the basis of our first prototypical implementation of the presented technique we use the SBMC tool LLBMC [SFM10] to complement deductive verification with VCC. Accordingly, our verification targets are C programs, and annotations are written in VCC's specification language. Nevertheless, the ideas presented here are not restricted to procedural programming languages like C, and can also be applied to programs written in object-oriented languages like Java or C++. Abstract types (interfaces), e.g., can be dealt with by providing suitable contracts for these interfaces or by giving a set of concrete instantiations. Dynamic typing can be taken into account by replacing method calls by case discrimination over the possible dynamic types. The fundamental problem of how to engineer suitable annotations for verification remains largely unchanged compared to procedural languages.

7.1. Preliminaries

7.1.1. Verification Targets

In the following, we consider the verification targets to be C programs, containing a set of function definitions. We consider these functions to be the modules of the program. We say that a C function f_A depends on a function f_B iff the function body of f_A (syntactically) contains a function call to f_B. This dependency relation, together with the set of functions of the system forms a directed graph. These graphs may contain arbitrary cycles depending on the implementation of the functions. In the following, for simplicity, we assume the graphs to be acyclic. In practice, mutually recursive functions would have to be specified together in one step and the verification methodology has to ensure that no cyclic reasoning occurs. For longer cycles in the call graph, techniques such as program slicing would allow us to split the graph and consider acyclic parts separately.

Similar to the notion of a root in a tree, we define as the roots in the dependency graph any node without a parent. The depth of a node is defined as the length of the longest path from this node to any of the roots in the graph. The set of all functions that a function f depends on is called $\mathit{children}(f)$; conversely, the set of all functions that depend on f is called $\mathit{parents}(f)$.

The depth of a node can be used to introduce a (topological) ordering $<$ on the nodes of the graph: $f_1 < f_2$ iff $\mathit{depth}(f_1) < \mathit{depth}(f_2)$. In the following, we identify functions with their corresponding nodes in the dependency graph, and we use the terminology of order theory for functions where appropriate.

7.1.2. Annotations and Their Semantics

In modular deductive verification, the specification $SPEC$ of a software system S is composed of the specifications of its modules (C functions) and data structures. Recall that $SPEC$ is a set of annotations, where each annotation consists of (a) one or more expressions of the specification language (pre-/post-conditions, invariants, assertions, etc.) and (b) the position of the annotation in the program, as stated in Definition 1 on page 79. We further assume that each annotation is local to a single function of the specified system, i.e., $SPEC$ is the disjoint union $SPEC = SPEC_1 \cup \ldots \cup SPEC_n$ of specifications $SPEC_i$ for each of the functions f_i of which S consists.

Since the specification languages we consider are modular, we have

$$S \vDash SPEC \quad \text{iff} \quad f_1 \vDash SPEC_1, \ldots, f_n \vDash SPEC_n \ .$$

Also, as before (Definition 4 on page 80) the relation $\vDash$ is monotonic w.r.t. adding annotations:

$$S \vDash SPEC \cup SPEC' \ \text{ implies } \ S \vDash SPEC \ .$$

This monotonicity condition requires, for example, that a precondition is not considered to be an annotation on its own but only in combination with a post-condition. Adding a lone precondition may weaken a specification while adding a pre-/post-condition pair always strengthens it.

As explained in Chapter 5, annotations both consist of auxiliary and requirement annotations – in our terminology, $SPEC$ covers both types of annotations, thus in the following $SPEC$ is a synonym for $REQ \cup AUX$.

7.1.3. The Verification Task

Given a software system S, consisting of the functions $f_1, \ldots, f_n$ and a requirement specification REQ_S, such that $S \vDash REQ_S$, the task of the user is to find a set of auxiliary annotations AUX_S in the sense of Definition 7 on page 82, s.t. $S \vdash REQ_S \cup AUX_S$.

We tacitly assume that an already proved auxiliary annotation $a \in AUX_S$ can be used in subsequent proofs for other annotations in REQ_S and AUX_S. Thus, by using such auxiliary annotations as *lemmas*, proofs for elements of REQ_S may be greatly simplified or may even become possible at all in a given verification system.

Note that both REQ_S and AUX_S are composed of specifications for each of the functions f_i in S, i.e.,

$$REQ_S = REQ_1 \cup \ldots \cup REQ_n \ \text{ and } \ AUX_S = AUX_1 \cup \ldots \cup AUX_n \ .$$

Assuming soundness of the verification system,

$$S \vdash REQ_S \cup AUX_S \ \text{ implies } \ S \vDash REQ_S \cup AUX_S \ ,$$

and due to the requirement that $\vDash$ is monotonic w.r.t. adding annotations, a solved verification task (i.e. $S \vdash REQ_S \cup AUX_S$) implies $S \vDash REQ_S$.

If, on the other hand, S does not satisfy REQ_S, the verification task has no solution, i.e., no appropriate AUX_S exists. In that case, the verification system may still give the user feedback that helps to correct the requirement specification and/or the implementation.

7.1.4. The Modular Verification Process

The set of annotations of a function f consists of two parts: One part can be used in the correctness proof for calling functions of f (e.g., pre-/post-conditions of f), while the rest can only be used in the verification of the function f itself (e.g., loop invariants). We call the former set of annotations the *external* specification of f, while the latter is named the *internal* specification of f. Which kind of annotation belongs to which of these categories is determined by the verification methodology built into the verification tool and the verification task at hand.

When verifying a function f using a modular verification approach, the external specifications of the children f' of f are used in the correctness proof of f instead of their implementation. Thus, the external parts of the auxiliary specification AUX' of f' are not only relevant for the verification of f' but can also be used as lemmas in the proof of other functions, which in some sense breaks the modularity of the verification process. There exist dependencies between the auxiliary annotations for the different functions, which makes finding a complete set of auxiliary annotations to solve a verification task a difficult problem.

7.1.5. Top-down and Bottom-up Verification

The user of a deductive verification system may chose different orders in specifying and verifying the modules of a system. The extreme cases are:

Top-down verification The process starts with specifying and verifying the top-level functions with minimal depth in the dependency tree, before proceeding to verify functions with greater depth.

Bottom-up verification The process starts with specifying and verifying functions with maximal depth (leaves in the dependency tree) and proceeds to functions with smaller depth.

In an ideal world, given a prover that never fails due to time-outs, the modular software verification process would proceed top-down, starting with the requirement specification of a top-level function f. All children of f are then specified using the strongest possible contract (which by definition must be sufficient if any annotation is sufficient). Then, f can be proven to be correct with the help of auxiliary internal annotations given by the user. The process repeats with the children of f, until all functions of the system are verified to be correct w.r.t. their specifications. Similarly, this process could also be performed bottom-up – the user annotates each leaf in the dependency tree with its strongest possible contract. This contract is always sufficient to prove any parent function correct w.r.t. its specification (if there is no bug in the program or specification). Again, the annotation process is repeated until all maximal functions have been verified to be correct w.r.t. its specification.

Unfortunately, using the strongest contracts is not a good idea in practice. They are (a) hard to find and (b) hard to prove. So, in practice, the solution to a verification task is a set of auxiliary annotations that are just strong enough. To support the user in the process of finding a solution and making the search less chaotic is the goal of the work presented.

7.1.6. Bounded Software Verification

So far, the verification tools VCC and KeY introduced in this thesis perform “full” deductive verification in a sense that the methods guarantee correctness independent of the size or concrete values of program inputs. A different approach is taken by *bounded verification*, providing correctness statements only up to a fixed size of data structures used in the program or number of loop iterations taken. To compare both verification methods, in the following, we apply them on a small sample program. In the subsequent section, we will then introduce one implementation of the bounded verification technique used in our case study, the SBMC tool LLBMC [SFM10].

For this, consider the program to compute the sum of the first n integers plus a constant c (both c and n are input parameters), as shown in Figure 7.1a. Besides the actual program (in black), we label (in gray) properties of the program state during program execution. The property $n \geq 0$ is the precondition of the program and $x = (n^2 + n)/2 + c$ its postcondition.

```
$n \geq 0$
int i = 0
int x = 0
while (i < n)
{
  i++
  x += i
}
x += c
$x = \frac{n^2 + n}{2} + c$
```

(a) Original program

```
$n \geq 0$
int i = 0
int x = 0
while (i < n)
inv $i \leq n$
$\wedge\ x = \frac{i^2 + i}{2}$
{
  i++
  x += i
}
x += c
$x = \frac{n^2 + n}{2} + c$
```

(b) Program with annotated loop invariant

```
$n \geq 0$
int i = 0
int x = 0
if (i < n) {
  i++
  x += i
  if (i < n) {
    assume false
  } else goto out
}
out: x += c
$x = \frac{n^2 + n}{2} + c$
```

(c) Program with loops unrolled once

Figure 7.1.: Reasoning about functional correctness of a sample program

With full verification, we are able to prove that the program adheres to its pre- and postcondition pair (e.g., using weakest precondition computation, as explained below). For the unbounded case, the user needs to supply information describing the effect of executing (a variable number of) loop iterations in the program via loop invariants. For our example program, the appropriate invariant is shown in Figure 7.1b. Finding the right loop invariant for this example is trivial – for real-world implementations, however, this is a difficult and sometimes infeasible task, as there might not even be a suitable abstraction in form of a concise invariant that describes the loop’s effects.

```
1  (0 < n → ((1 < n → true)  ∧ (1 ≥ n → 1 + c < 9)) ∧
2  (0 ≥ n → 0 + c < 9)
3  int i = 0; int x = 0
   (i < n → ((i + 1 < n → true)∧
5           (i + 1 ≥ n → x + (i + 1) + c < 9)) ∧
6  (i ≥ n → x + c < 9)
7  if (i < n) {
     (i + 1 < n → true) ∧ (i + 1 ≥ n → x + i + c < 9)
9    x += i; i++
10   (i < n → true) ∧ (i ≥ n → x + c < 9)
11   if (i < n) {
       false → x + c < 9 ⇔ true
13     assume false
14     x + c < 9
15   } else goto out
   }
   x + c < 9
18 out: x += c
19 x < 9
```

Figure 7.2.: Weakest precondition computation for property: $x < 9$

Another option which does not need further user-supplied information is to use bounded verification. Loops in the program are dealt with by examining program executions only up to a few loop iterations (the *bound*). This constraint on the length of execution paths allows us to transform loops into if-cascades – as depicted in Figure 7.1c for executions containing at most one loop iteration. The assume statement inserted by the transformation in the second if-block is used for weakest precondition computation: independently from the properties that actually hold at this point in the execution, this statement adds *false* as an assumption so that from this point on, *any* property holds, effectively treating all executions of the original program that pass this point as if they unconditionally establish the postcondition.

Starting from either the original program with annotated invariants or the unrolled program, the next step in showing correctness of the program is to generate the weakest precondition for the given postcondition w.r.t. the program, resulting in a first-order logic formula. The weakest precondition is a property of program states s.t., if the program is started in a state satisfying the weakest precondition, then it terminates in a state that satisfies the postcondition. If the actual precondition of the program implies the weakest precondition, then the program is correct w.r.t. the given pre-/postcondition pair. Whether this implication holds is typically checked automatically using an SMT solver.

For our example program, Figure 7.2 shows the intermediate properties resulting from weakest precondition computation for a simple postcondition P: $x < 9$. For P to hold after execution of the final statement `x+=c` in line 18 of the program, P with all occurrences of x replaced by $x + c$ has to be true beforehand. Corresponding rules for the other statement types are applied to all statements of the program to get the weakest precondition of the program w.r.t. P, as seen in lines 1-2 in Figure 7.2. The constructed weakest precondition is equivalent to $(n = 1 \rightarrow c < 8) \wedge (n \leq 0 \rightarrow c < 9)$.

As explained, bounded verification only guarantees correctness up to a given number of loop iterations. Thus, for parameters that affect the number of loop iterations in the program execution (in our example: n), we only get correctness results for a subset of values. In the weakest precondition for our example program in Figure 7.2, the subformula $(1 < n \rightarrow \mathrm{true})$ captures this: for any concrete value for n greater than one, the whole formula evaluates to true, although the postcondition $x < 9$ actually may not hold.

7.1.7. The Low-Level Bounded Model Checker (LLBMC)

One instance of the SBMC technique is implemented by LLBMC, a static analysis tool to discover errors in programs for finite execution traces, developed at the Karlsruhe Institute of Technology. It supports sequential C/C++ programs, using a flat, byte-level memory model to faithfully handle semantics of low-level C operations. Program properties that can be checked for include (a) built-in correctness specifications like absence of integer overflows or correct reference to memory (e.g., only allocated memory is accessed) and (b) user-defined specifications similar to C-assertions extended by LLBMC-specific primitives, e.g., to assign arbitrary values to variables of primitive type. As only bounded program runs are considered and the state space of the program is finite, the problem of determining whether the program meets its specification is decidable.

LLBMC translates the C code annotated with the assertions and assumptions to a quantifier-free first-order formula (over decidable background theories) which is unsatisfiable iff the C program is correct w.r.t. the (implicit and explicit) specifications for all execution traces up to a given bound. This formula is passed on to the SMT solver STP [GD07]. Any model found by STP indicates a specification violation and the assignments of values to the variables in the model can be mapped back to execution traces of the original program.

The implementation of LLBMC separates concerns by using an intermediate representation of the problem, similar to the tool-chain architecture of VCC: Transformation of the program to a first-order formula involves a translation into LLVM's [LA04] intermediate single static assignment form (IR) of the source code. Unlike VCC, the first translation step from C to LLVM-IR is generic in that it uses the CLANG [CLANG] compiler front-end to handle parsing the C code and IR code generation. This implies that any feature to be included on the C code level for use with LLBMC (e.g., further annotation types) either has to fit into the syntactic structure of the C language, respectively CLANG grammar, or requires a modification of the CLANG compiler front-end.

On the level of the LLVM intermediate representation, the C program to check is translated to a program with only finite execution traces by inlining method bodies and unrolling loops. For some loops, the number of loop iterations is small (depending on the input parameters of the program etc.), so these loops can be fully unrolled to provide a complete analysis for this part of the implementation – LLBMC includes a heuristic to determine this number of loop unrollings if the loop condition contains the loop range in a syntactically accessible manner. If this heuristic fails, the user has to provide a default for the maximum number of loop unrollings.

As another intermediate stage between the LLVM-IR representation and the SMTLIB-formula, a custom LLBMC intermediate logical representation is used on which simplification steps through term rewriting are performed, before a formula in the QF_AUFBV[1] fragment is generated and passed to the SMT solver.

7.2. Deductive Verification of Large Software Systems

Recall that in modular verification a function is verified using the external specifications of its children. If the verification of a function f succeeds, then that does not imply that its children are correct w.r.t. their specifications. In case a child function f' does not satisfy its specification, there may or may not be a different auxiliary specification for f' that is both satisfied by f' and sufficient to verify f.

Similarly, the external specifications of a child f' may be insufficient to verify f. Again, there may or may not be an alternative specification for f' that solves the problem.

When adhering strictly to a bottom-up verification process, one will never encounter the case that one of the children does not satisfy its specification, but it may very well happen that the specification of a child is insufficient to verify its parent. On the other hand, when verifying top-down, one will never end up with insufficient specifications of the children, but a specification of a child f' that is not satisfied by f' may very well occur. That is, independently of the order in which functions are specified and verified, one of the two problems remains.

Even always using the strongest possible contract for all functions is not an option here: while providing a stronger contract for a function f' may help in the verification of the parents of f', it also makes verifying f' more difficult. In practice, the user has to provide a specification for f' that is strong enough to verify all parent functions of f' and weak enough to verify f' itself.

Moreover, the logical strength of a contract is not its only relevant property but its syntactic form is just as important. As it is hard to foresee which specification of a function f is the most appropriate without paying attention to all call sites in the parents, in practice, neither a strict top-down nor a strict bottom-up approach is applied. Instead, during the process a continuous adaptation of the specification takes place during which the specifications of calling and called functions are changed in alternation until verification of the software as a whole succeeds – this process is shown in Figure 7.3a on page 123. A further possibility, which is not shown in the figure, is that the verification process fails because the implementation does not satisfy the requirement specification (in which case refining the annotations cannot help). Then, the implementation and/or the requirement specification need to be changed and the verification process restarted. The iterative process shown in Figure 7.3a is often applied locally, i.e., only one pair of caller and callee is considered at a time. As other functions may also use the callee and depend on its contract, changes in this contract may have to be propagated to various other parts of the system.

[1]Quantifier-Free first-order logic with the theories of Arrays, Uninterpreted Functions and BitVectors

7.2.1. Object Orientation

Our proposed approach may also help with specifying and verifying object-oriented programs. In the following, support of our method for two particular features of OOP is examined, namely polymorphism (through class based inheritance with method overriding) complying with the Liskov substitution principle, as well as class invariants.

Polymorphism. For this, consider a class A implementing a method m and n subclasses of A called $\mathrm{B}_1, \ldots, \mathrm{B}_n$, each overriding A's implementation of method m.

Regardless of whether specifying bottom-up or top-down, when specifying the contract of the method m in A, two problems may occur. One is concerned with the issue described previously: to come up with the contract for A.m, the user has to consider all call sites of A.m. However, in our object-oriented setting, also all overridden implementations (or contracts) of A's subclasses $\mathrm{B}_1, \ldots, \mathrm{B}_n$ have to be taken into account. For each B_k, the precondition of A.m has to imply the precondition of B_k and the postcondition of B_k has to imply A.m's postcondition to satisfy the behavioral subtyping principle.

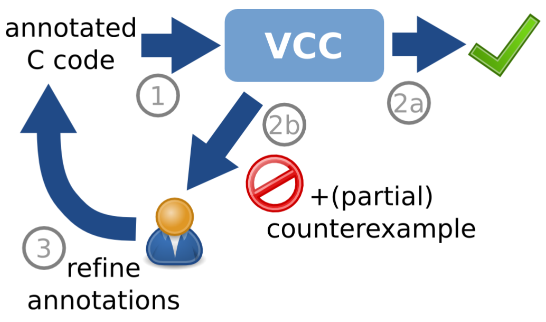
The last problem can be mitigated by using a technique called *lazy behavioral subtyping* described by Dovland et al. [Dov+10]: instead of using the contract of A.m as a constraint for the contracts of m in subclasses, only the properties of A.m that are actually used at the call sites in the program are required to be fulfilled by subclasses of A. To generate those properties, requirement specifications of the methods containing calls to A.m are required. In contrast, our proposed approach focuses on an earlier state in the specification process, where most requirement specifications may not yet be available.

Already while the user specifies A.m, our method would be able to give an indication whether the contract of A.m is sufficient for all call sites of A.m, as well as whether the contract is compatible to the implementations of all derived classes. The latter is implemented by inserting the appropriate method bodies of derived classes for calls to A.m in several runs of our tool. Likewise, in a bottom-up specification approach, after specifying m in any of A's derived classes B_k, this contract can be checked against the call sites and implementations of the superclass A, as well as the siblings B_j. A successful check against the siblings of B_k suggests that the contract of B_k is also a suitable contract for the implementation of the method in the superclass.

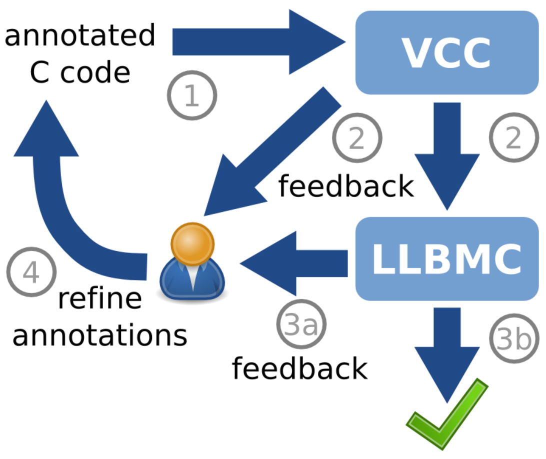
Class invariants. Another means to express functional properties of object-oriented programs are class invariants. Although the C programming language has no concept of objects, in VCC, structured data types are treated similarly and also can be annotated with invariants, as already explained in Section 2.4.1.

For our purposes, only two states such an object may have according to the VCC methodology are relevant: it is either open or closed. Invariants of a closed object are known to hold throughout the execution of a (sequential) program, until it is opened, which is a prerequisite to being able to modify it. Establishing invariants in this methodology is only needed when closing an object.

Our proposed method could be extended to handle class invariants in the scope of the VCC methodology as follows: if the locations in the program are known where objects



(a) Normal VCC workflow



(b) Counterexample guided *manual annotation* refinement (CEGMAR)

Figure 7.3.: Comparison of regular VCC workflow with our CEGMAR process.

are being closed, checking invariants is reduced to checking an assertion (of the same property) at these locations. To identify these locations in the program is non-trivial in general, but for simple cases this could be approximated with the help of heuristics, e.g., always opening (closing) objects at method boundaries; or before (after) the first (last) modification to the object in a method. In cases where an invariant check fails due to wrongly placed open (close) annotations, the user may annotate the program with open (close) statements, overriding the automatically generated annotations.

7.3. The Integrated Verification Process

In the following, we describe how to integrate software bounded model checking into the annotation-based verification process, thereby taking advantage of the strengths of both methods. As said above, we use the tools VCC and LLBMC to illustrate our approach.

The central idea of our method is to use SBMC to support the process of finding a set of auxiliary annotations, AUX, for a given system S that allows the deductive verification tool to prove that S satisfies its requirement specification REQ. The resulting integrated process is illustrated in Figure 7.3b. The name CEGMAR is inspired by the counterexample-guided abstraction refinement (CEGAR) technique [Cla+00] used in model checking – though in CEGMAR annotations are refined instead of abstractions.

Note that we do not use the term refinement in its strict mathematical meaning here. Instead, we have a more colloquial interpretation in mind, where refinement simply means iterations towards a specification which is fit for its purpose. Also, this kind of refinement contains a manual component, which makes CEGMAR a machine-supported verification process, not a fully automatic algorithm.

CEGMAR aims at finding suitable auxiliary specifications for the full system S, but at any given point in time, some function f is in the focus of the process. The process starts from the given requirement specification REQ_f for a function f and a (possibly empty) set AUX_f of auxiliary annotations.

In Step 1 (Fig. 7.3b), the annotated C code relevant for proving f correct w.r.t. its (requirement and auxiliary) specification is passed to VCC for verification. The result of VCC's verification attempt for f is given to LLBMC (Step 2 in Fig. 7.3b). Then, in case both VCC and LLBMC agree that f satisfies its specification, the refinement-loop for f ends successfully (Step 3b). After this, some other function moves into focus or, if all functions have been verified, the verification task has been successfully completed.

Otherwise, if one of the tools (or both) fails to verify f, the user has to refine some auxiliary annotations (Step 4), using the feedback of VCC and LLBMC (from Step 2 respectively 3a). After refining the auxiliary annotations in Step 4, the next iteration starts with Step 1. If changing the auxiliary specifications is not sufficient according to the feedback from the tools, i.e., there is a problem in the implementation or the requirement specification, then the refinement loop for f terminates and can only be restarted after the implementation and/or the requirements have been fixed.

In Step 2, using VCC, the correctness of f is only proven *locally*, i.e., the external specifications for $children(f)$ are used without checking that they are satisfied. Feedback from VCC is either (a) the statement that f is correct w.r.t. its specification or (b) a list of annotations that cannot be proven (possibly together with counterexamples). In contrast, LLBMC is used in our integrated approach to check correctness of a function f *globally*, i.e., the implementation of all functions called by f (directly or indirectly) is taken into account. We identified three different properties to be checked with LLBMC:

A. f satisfies its specification;

B. all functions in $children(f)$ satisfy their external specifications;

C. the precondition of f holds at all points where f is called in $parents(f)$ (invocation contexts).

Each of the three checks A–C has three possible outcomes: either (a) the property in question holds up to a certain bound on the length of traces, or (b) LLBMC provides an error trace falsifying the property, or (c) there is a time out.

The three checks differ in which part of the implementation and annotations are given to LLBMC, as well as the consequences of the check for the verification process, as described in the following.

A: Checking that f satisfies its specification. The implementations of f and all descendants of f are passed to LLBMC for model checking. The implementation of f is checked w.r.t. all annotations of f that VCC reports to be violated. LLBMC can provide the user with feedback on which unproven specifications are indeed violated by the implementation and which are likely satisfied (because no counterexample was found within the given bound), but just not provable by VCC without refining the annotations.

Even if VCC could verify that f is correct (based on the external specification of functions called by f), LLBMC is still applied. This allows one to discover cases where VCC's correctness proof for f only succeeded due to an erroneous external specification of a child of f.

B: Checking that child functions satisfy their external specification. For this, the implementations of all descendants of f are passed to LLBMC for model checking. The functions in $\mathit{children}(f)$ are checked to satisfy their external specifications. This check helps to rule out correctness proofs for f that are erroneous because they rely on faulty specifications of f's children.

C: Checking Invocation Contexts. For each function $g \in \mathit{parents}(f)$, the implementations of g and all descendants of g are passed to LLBMC. Here, the property to check is the pre-condition of f. Checking the invocation contexts helps to avoid writing specification for f that cannot be used in the proofs for other functions in the system.

Note that in all these cases, LLBMC is not used to check whether a function satisfies an annotation in general, but to check that the function satisfies the annotation in the context in which it is called. The context may be defined by the function's precondition or by the context in one of its parents.

The benefit of the proposed integration of SBMC into deductive verification results mainly from the fact that SBMC is not modular. During the verification of a function f, LLBMC uses the implementation of the children and parents of f instead of (only) using the external specification of the children as VCC does. Because of this difference, LLBMC and VCC can provide the user with different information about the functions and their annotations (e.g. counterexamples). For the verification process, the information provided by LLBMC is a valuable addition to the information provided by VCC.

Translation from Annotated C to LLBMC Input To get meaningful feedback from LLBMC for the correctness of C code annotated with VCC specifications, we have to emulate VCC's specification constructs as far as possible, at the same time taking into account the different semantics of specification satisfaction underlying the bounded, whole-program analysis tool LLBMC.

More specifically, for a given program P with specification SPEC, we have to produce a modified program P', together with annotations SPEC' in the annotation language of LLBMC (with the help of a translation function $t(P{+}\mathit{SPEC})$), s.t. the result of LLBMC on $P'{+}\mathit{SPEC}'$ allows to infer properties of the original annotated implementation.

Firstly, this requires our approach not to produce any false positives, i.e., a counterexample for $P'{+}\mathit{SPEC}'$ produced by the bounded checking tool B (for a given number m of method inlining depth and l of number of loop unrollings) indicates that the original program does not satisfy its specification. Formally:

$$\text{if } \vdash_{B(m,l)} \neg(P'{+}\mathit{SPEC}') \text{ then } \nvDash P{+}\mathit{SPEC}$$

As this requirement is fulfilled by the trivial implementation that simply always returns success, we also would like to exact a notion of completeness of our method. However, the converse of the property stated above does not hold, i.e., our approach is necessarily incomplete w.r.t. the specification satisfaction relation $\models$ given for the deductive verification tool, for several reasons:

1. LLBMC only applies to bounded execution traces whereas the deductive verification approach imposes no such restriction on program execution.
2. Our approach is not able to handle all specification features supported by VCC (e.g., quantification over unbounded domains would lead to the correctness proof being undecidable and is not compatible with our goal of having a push-button tool).
3. LLBMC does not support modular verification – some annotations have a different semantics in the case of whole program analysis (e.g., postconditions of functions in bounded verification have to hold in all execution traces from the main entry point of the analyzed module, compared to modular deductive verification, where all executions allowed by the – possibly more liberal – precondition have to fulfill the postcondition).

Instead, our approach is complete w.r.t. a modified specification satisfaction relation $\models_B$ (i.e., with a changed programming language semantics for execution of while loops and method invocations, as well as a different semantics for some annotation features).

Note that we only consider sequential C programs, i.e., our approach does not support volatile variables or the associated specification features (e.g., two-state invariants or claims). Also, we assume that the pair P+$SPEC$ is well-formed, i.e., both the program and annotations are syntactically correct but also VCC's requirements regarding ghost code not influencing execution of the program are met.

We perform translation of the annotated program P+$SPEC$ in a way that maintains the behavior of P in the sense that traces of the translated program P' and P are identical (modulo stuttering of P) on the state of P – i.e., all variables used to store intermediate results of the evaluation of an assertion in P' have to be disjoint from the set of variables used by P. In addition, P is modified only by introducing new statements that must not interfere with execution of P, similar to the requirements for ghost statements in VCC (e.g, the added code has to terminate).

Conversion of VCC annotations in the smaller vocabulary of LLBMC can be divided into two parts: substituting the different VCC specification statements (like `requires` or `invariant`) by simple assertions and assumptions, retaining the original expressions asserted/assumed by the specification statements. This translation step takes place on the VCC level and produces a pair P'+$SPEC'$ that is equivalent to the original annotated program P+$SPEC$. In a second step, the semantics of expressions remaining in VCC syntax from step one have to be emulated using C statements for use with LLBMC (e.g., as LLBMC does not support quantifiers or specification keywords like `\old` or `\result`).

Most of the translations in the first step are straightforward:

Method contracts Postconditions of functions are replaced by assertions either just before each exit point in the function, or after all call sites of the function. Preconditions, in contrast to modular verification, do not serve as assumptions for subsequent assertions (e.g., the postcondition of the function), but for whole program analysis are always treated as an obligation for the preceding code to establish a state satisfying the precondition. Hence, preconditions are also translated to assertions, positioned as the first statement(s) in the method body.

Loop invariants The informal semantics of loop invariants is that the property given by the invariant holds initially before executing the first loop iteration and that this property is preserved through all loop iterations. This can be equivalently expressed by using two assertions – one before the loop and one as the last statement of the loop body (assuming absence of goto or break statements). Whereas in modular verification the loop invariant, together with the rest of the program, is used to infer the postcondition, for checking specifications with LLBMC, we do not include the invariant property as an assumption for subsequent annotations.

Data structure invariants Finding appropriate states in the program a data structure invariant is supposed to hold can at best be approximated (see Section 7.2.1). Given annotations for wrapping and unwrapping C structures by the user, checking the invariants is done by both asserting the corresponding property directly after unwrapping and immediately before wrapping the structure.

Ghost code Aside from specification-only data types (like maps), ghost code can safely be included without the ghost modifier in the translated program, given our non-interference assumption stated previously.

This list of specification statements is incomplete, e.g., support for the `writes` clause is missing, which would incur non-trivial bookkeeping overhead to track the memory regions allowed to write to according to the VCC ownership methodology. Instead, we rely on the built-in checks for validity of memory accesses provided by LLBMC as a first approximation of admissible changes to memory. Providing further translations beyond the annotations given in the list above is left for future work – any annotation type not supported is simply omitted in the translation for now.

Concerning VCC's specification expressions, one notable extensions over pure C expressions are quantifiers. Handling quantified formulas in our approach is restricted to simple cases that correspond to bounded quantification, i.e., the range of the bound variable is restricted to a (small) finite domain – the truth value of the quantified expression can then be evaluated by enumerating all instances of the expression for each of the assignments to the bound variable, similar to the technique described by Zee et al. [Zee+07].

To determine feasibility of an automatic translation, we developed a prototypical implementation to generate the `assert` and `assume` statements for LLBMC, and used it in the evaluation described in Section 7.5. Figure 7.4 shows an exemplary excerpt from the VCC specification of function `copyNoDuplicates` (from the example in Section 7.4) and the result of a manual translation of that specification into LLBMC input. We expect to be able to estimate better how well translation of specifications is possible in general, and to what degree it can be automated, once our prototype has matured and once LLBMC comes closer to VCC expressiveness in specification.

```
//no 'new' items in result
_(ensures \result != NULL ==>
  ∀ uint i; i < \result->count ==>
    (∃ uint j; j < ↙
       ↳ source->count ∧
      \result->items[i] ==
        source->items[j]))
```

```
cnt = result->count;
//no 'new' items in result
if (result != NULL)
  for (i = 0; i < cnt; ++i) {
    int found = 0;
    for (j = 0; j < cnt; ++j)
      if (result->items[i] ==
          source->items[j])
        found = 1;
    assert(found == 1);
  }
```

Figure 7.4.: Exemplary excerpt from the requirement specification of the function `copyNoDuplicates` (left) and its translation into LLBMC input (right).

7.4. A Typical Specification Scenario

In the following we present an example that demonstrates the issues of modular verification mentioned before and how integration of software bounded model checking into the verification process can help attenuate these.

7.4.1. The Program to be Verified

Consider the following C data structure implementing a queue data type:

```
typedef struct queue_t {
  int *items;
  int count, capacity;
} queue, *pQueue;
```

Here, `count` denotes the number of items in the queue and `capacity` the fixed size of memory that has been allocated to store all items of the queue. In our case, these items are integers and are stored in the array that starts at the memory address `items`.

The top-level function we want to verify is `copyNoDuplicates` (see Listing 7.5), but in total there are three functions involved in the verification process:

- `pQueue copyNoDuplicates(pQueue dst, pQueue src)`
  Given a queue dst and a queue src, this function modifies dst so that it is a copy of src, except that duplicate elements of src occur only once in dst.
- `pQueue initQueue(int capacity)` (not shown)
  Allocates memory for a queue structure as well as the appropriate amount of memory for storing `capacity` number of items. It also initializes the queue data structure to correspond to the empty queue. This function is called by `copyNoDuplicates`.
- `void insert(pQueue q, int val)` (shown in Listing 7.5)
  Inserts an item val into a queue q so that if the queue is in ascending order, it remains ordered. This function is called by the implementation of `copyNoDuplicates`.

```c
pQueue copyNoDuplicates(pQueue dst, pQueue src) {
  dst->count = 0;
  for (int i = 0; i < src->count; i++) {
    int sVal = src->items[i];
    int j = 0, contained = 0;
    while(j < dst->count && dst->items[j] <= sVal) {
      if (dst->items[j] == sVal) {
        contained = 1;
        break;
      }
      j++;
    }
    if (!contained) insert(dst, sVal);
  }
  return dst;
}

void insert(pQueue q, int val) {
  if (q->count == q->capacity) return;
  int i,j;
  for (i = 0; i < q->count && val > q->items[i]; i++) {}
  for (j = q->capacity-1; j>i; j--)
    { q->items[j] = q->items[j-1]; }
  q->items[i] = val; q->count++;
}
```

Listing 7.5: Implementation of `copyNoDuplicates` and `insert`

There are two peculiarities about this implementation of `copyNoDuplicates` that the verification engineer might not be aware of:

1. The implementation relies on `insert` retaining sortedness of the queue `dst`. This is because the algorithm stops searching for a matching element as soon as a greater element is encountered. Note that sortedness is not an invariant of the queue data structure, so queue `src` may be unsorted.
2. Inserting into a queue fails silently if the capacity of the queue is reached.

In the following, we will use the example to show why a user who is not supported by software bounded model checking will have trouble identifying these problems during verification, independently of whether a top-down or a bottom-up approach is chosen.

7.4.2. Local Verification of `copyNoDuplicates`

Additional to the issue of interdependent method contracts, local verification of one method against its specification is often already non-trivial. Exemplary for the amount of annotation overhead and user effort needed for verification, we will examine proving method `copyNoDuplicates` correct against a given specification that is known to be fulfilled by the implementation – to separate issues in local verification from the difficulties in finding suitable method contracts of called methods, which is the content of the subsequent sections, we assume given a suitable contract for the method `insert`.

The external specification of both methods relevant for our example is shown in Listing 7.6, respectively Listing 7.7. Note that the contract of `copyNoDuplicates` only describes a single aspect of the functional correctness of the implementation (which is adequate for the demonstration purpose) – in the same way, the contract for `insert` (as shown in Listing 7.6) is not the strongest possible specification, but one suitable to verify `copyNoDuplicates`, e.g., the postcondition of `insert` is missing the property that the resulting array remains sorted.

As insertion of an element into a queue may fail if the queue is full, the postcondition of `insert` consists of two parts: (1) full capacity of the queue is reached, so the queue remains unchanged (in particular, neither field count nor any elements in the queue are changed) and (2) there is at least one free array element in `items` and (2a) all elements in the queue before are still contained in the queue, (2b) all elements in the resulting queue were in the queue to begin with or are equal to `sVal`, (2c) the value `sVal` has been added to the queue and (2d) count is increased by one. In both cases, the capacity of the queue does not change and `insert` ensures that the queue is wrapped after method invocation. Explicitly mentioning equality of the fields of parameter `q` that are unchanged allows us to use the simple writes clause containing only `q`.

The goal of the verification task demonstrated here is to verify the single postcondition of function `copyNoDuplicates`, as given in Listing 7.7, stating that all elements in the queue returned by the function are taken from the queue `source`.

The amount of auxiliary annotations needed to verify the implementation, measured in lines of specification, is almost half of the amount of the implementation, as shown

```
int insert(pQueue q, int val)
  _(maintains \wrapped(q))
  _(ensures \same(q->capacity))
  _(ensures \old(q->count) == \old(q->capacity) ==>              //(1)
        \same(q->count)
     && \forall uint n; n < q->count ==> \same(q->items[n]))
  _(ensures \old(q->count) != q->capacity) ==>                   //(2)
     (\forall uint j; j < \old(q->count) ==>                     //(2a)
        \exists uint k; k < q->count
          && \old(q->items[j]) == q->items[k])
      && (\forall uint j; j < q->count ==>                       //(2b)
            (\exists uint k; k < \old(q->count) && q->items[j] ==
            \old(q->items[k])) || q->items[j] == val)
      && \exists uint i; i < q->capacity                         //(2c)
                           && q->items[i] == val
      && q->count == \old(q->count) + 1)                         //(2d)
  _(writes q)
{ ... }
```

Listing 7.6: Partial method contract for function `insert`

in Listing 7.8 (with 10 lines of specification and 22 lines of code). For the verification of insert, the amount of work further shifts towards the auxiliary annotations, with a ratio of one line of annotation per line of implementation. In addition, for the sake of a concise presentation, the annotations have been minimized after a version of the method with more elaborate annotations needed to come up with the final proof has been verified.

A simple metric like lines of code is not meaningful on its own, rather the complexity of the auxiliary specification has to be taken into account. In this regard, most of the annotations seem straightforward, at least in retrospect – finding the right lemmas, e.g., the assertions in line 30 to line 35 in Listing 7.8 on page 133, is a common problem and in this case deserves further explanations because it demonstrates the issue of finding appropriate triggers as explained in Chapter 4.

```
pQueue copyNoDuplicates(pQueue source)
  _(maintains \wrapped(source))
  _(requires source->count < INT_MAX - 1)
  _(ensures \result != NULL ==>
      forall uint i; i < \result->count ==>
        \exists uint j; (j < source->count
          && \result->items[i] == source->items[j]))
{ ... }
```

Listing 7.7: Method contract of function `copyNoDuplicates`

The outer loop of the implementation of `copyNoDuplicates` ranges over the elements of the `source` array, checking whether the element is already contained in `dest` using the inner while loop. Only if the element is not yet contained in the `dest` array, it is inserted into `dest`. The assertions in line 30 to line 35 help to establish knowledge about the contents of the arrays `dest` and `source` in the latter case, i.e., after the call to `insert` was taken – only both assertions together allow Z3 to establish the loop invariant after the loop body execution. To relate the contents of the `source` array before the call to `insert` to the program state at line 30, a ghost variable of type `\state` is introduced which provides a reference to the current state at line 25. Expressions can be evaluated in a specific state using the `\at` function, as seen in line 30, where we assert that `insert` does not modify the `source` array (as the contract for `insert` states that at most the formal parameter `q` is written to and the actual parameter `dest` is disjoint from `source`).

Both contracts for functions `insert` and `copyNoDuplicates`, besides describing only some aspects of the functional correctness of the implementation, are one possible formalization amongst some alternatives – in particular, this specification is suitable for the verification with LLBMC described further. To simplify full functional verification, the contract of `insert` could, e.g., make use of a ghost out-parameter, providing the caller of `insert` the index into the source queue where the value `sVal` has been inserted – removing the existential quantification in the contract of `insert`.

7.4.3. Global Verification of `copyNoDuplicates`

Applying the two different verification orders introduced in Section 7.1.5 to our running example results in the following concrete verification steps:

Bottom-up Verification When verifying bottom-up, the first two functions that are to be verified correct in our case are `initQueue` and `insert`. Because we are mainly interested in interaction between `copyNoDuplicates` and `insert`, from now on we assume that `initQueue` has been verified and does not need to be considered further.

The second issue mentioned in Section 7.4.1 (finite capacity of the queue) is identified quickly with a bottom-up approach and therefore not further discussed. The first issue (sortedness) is considerably harder to identify, though.

Suppose that no requirement specification is given for `insert`, so any specification of `insert` is auxiliary. It is likely that the user correctly specifies that the item passed to `insert` is indeed inserted in the given queue. However, it is also likely that the verification engineer on the first try is not aware of the importance of sortedness and, consequently, the specification does not mention that insertion of elements retains sortedness of the queue. In that case, the specification of `insert` is not strong enough and verification of `copyNoDuplicates` will fail. But that is only noticed after `insert` has been successfully verified and the verification process has moved on to `copyNoDuplicates`.

Once `copyNoDuplicates` could not be proven correct, the verification engineer has to identify the cause for this. The too weak specification of `insert` is hard to spot, and the user may be tempted to believe `copyNoDuplicates` is not correctly implemented. The counterexample provided by VCC usually does not contain all necessary information to understand the issue.

```
pQueue dest = initQueue(source->count + 1);
if (dest == NULL) return NULL;

for (uint i = 0; i < source->count; i++)
  _(invariant \wrapped(dest))
  _(invariant \forall uint j; j < dest->count
              ==> (\exists uint k; k < source->count
                     && dest->items[j] == source->items[k]))
  _(writes dest)
{
  int sVal = source->items[i];
  uint j = 0;
  int contained = 0;

  _(assert dest \in \domain(dest))
  while(j < dest->count && dest->items[j] <= sVal)
  {
     if (dest->items[j] == sVal) {
       contained = 1;
       break;
     }
     j++;
  }

  _(ghost \state s3 = \now())

  if (!contained)
    insert(dest, sVal);

  _(assert \forall uint j; j < \at(s3, source->count)
        ==> \at(s3, source->items[j]) == source->items[j])
  _(assert \forall uint j; {\at(s3, dest->items[j])}
     j < \at(s3, dest->count)
        ==> (\exists uint k; k < source->count && \at(s3,
          dest->items[j]) == source->items[k]))
}
return dest;
```

Listing 7.8: Auxiliary annotations for function `copyNoDuplicates`

LLBMC on the other hand states, using Check A from Section 7.3, that function `copyNoDuplicates` does indeed satisfy its requirement specification and the relevant loop invariants – at least up to a certain size of the queue.[2] This indicates to the user that `copyNoDuplicates` is likely correct and the problem is either that the specification of `insert` is too weak or the auxiliary annotations are not enough to allow VCC to verify the property. This is an important cue towards the right direction and can therefore speed up the verification process.

Top-down Verification of `copyNoDuplicates` In a top-down verification approach, the top-level function `copyNoDuplicates` is the first to be verified. The first issue (the queue is sorted) is found early in the process, as verification of `copyNoDuplicates` will already uncover it. Instead, the second issue (queue has finite capacity) causes problems.

Consider a requirement specification of `copyNoDuplicates` consisting of an empty precondition and the following post-condition stating that all elements of the source queue are also contained in the resulting queue (this is only part of the actual requirement specification because it does not state that the result should not contain duplicates):

```
∀i; i ≥ 0 ∧ i < source->count  ⟹
    ∃j; j ≥ 0 ∧ j < \result->count ∧
        source->items[i] == \result->items[j]
```

In order to verify the implementation of `copyNoDuplicates` to satisfy this requirement, the user has to provide auxiliary specifications for the helper functions `initQueue` and `insert`. The verification of these auxiliary specifications is postponed in the top-down approach until after the contract of `copyNoDuplicates` is proven. Nevertheless they are already used in the proof for `copyNoDuplicates`.

If the user annotates `insert`, he/she may easily overlook the case where the queue has reached its capacity and insertion of yet another element fails (signaled by `insert` by returning an error code). Now, because of this omission, the specification of `insert` is too strong, which allows VCC to prove the contract of `copyNoDuplicates` – even though it is in fact not satisfied. Only when the verification of the contract of `insert` fails, this error is detected.

Then, after fixing the specification of `insert`, the verification engineer has to go back to `copyNoDuplicates` and re-verify that function, taking the modified specification of `insert` into account. In practice, the top-down approach results in numerous iterations until a function and all of its children are verified.

Using LLBMC can help resolve this issue early on, as LLBMC directly takes the implementation of `insert` into account, and not just its (too strong) specification. LLBMC uncovers the problem as soon as `copyNoDuplicates` is checked – even though VCC cannot detect any problem at this point.

[2]This size is determined by the bound applied during model checking.

7.5. Evaluation

Although the LLBMC-based bug-finding technique can be used even before the user has provided sufficient annotations to be able to run the VCC tool, short run times of the bounded verification tool are still important for our envisioned use case. The goal of our approach is to provide the user instant feedback in the style of automatic spell checking in word processors, whenever an annotation has been completed. While in some occasions, run times of several minutes might be acceptable, we aim for feedback within one minute, following the experiences of the VCC tool developers concerning turnaround times for verification [Coh+09b].

To demonstrate feasibility of the application of bounded model checking to verification problems specified for deductive verification tools, we built a first prototype which is able to translate C code with annotations for VCC to input for the LLBMC tool, as described in Section 7.3. The implementation of this translator reuses existing functionality of parsing and manipulating C source code in form of the CLANG compiler front-end for the LLVM compiler infrastructure. Based on the CLANG implementation, generating source code for VCC annotations is achieved by extending the visitors provided by CLANG working on an intermediate representation of C code as abstract syntax tree.

Applied to our running example introduced in Section 7.4.1, at the time of the FoVeOOS publication in 2011 [Bec+12], the performance of the LLBMC tool was not sufficient to provide a counterexample within the envisaged time frame of less than one minute, even for small problem instances. First experiments with other small examples, as well as using another bounded model checker (CBMC) seemed to confirm the underwhelming performance of the approach.

In the following, we will both report on repeating the experiments using a current version of the LLBMC tool in a more rigorous fashion in order to determine bottlenecks of our approach and demonstrate how to improve the implementation of our specification checking tool.[3] All experiments are based on the running example from Listing 7.5, checking the requirement specification of function `copyNoDuplicates` in two variants: (a) the destination queue is sufficiently large to store all elements of the source array, demonstrating performance of our approach when providing a correctness argument up to a given bound and (b) the implementation contains a bug, as the destination queue has not been allocated large enough to store all elements of the source – this version of the example shows how our tool performs when finding counterexamples.

Based on the small scope hypothesis [Jac06], we expect variant (b) to be a viable option in case checking the whole state space for a bounded correctness argument is infeasible.

7.5.1. Checking Correctness of `copyNoDuplicates` with LLBMC

Test Setup Checking the contract for `copyNoDuplicates` using our LLBMC-based specification checker requires to establish a program state that satisfies the preconditions of the function (e.g., both actual parameters for `source` and `dest` are non-null and

[3]Experiments are run on an Amazon EC2 computing instance of type r3.2xlarge with 61 GiB RAM, 26 Compute Units and a timeout of 2000 s. According to the EC2 documentation, "One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor."

memory for at least `capacity` elements is allocated at memory location pointed to by the field `items`). As LLBMC is a whole-program analysis tool, the usual method to establish such a well-formed memory state is for the tool user to provide an (abstract) initialization function.

```
for (int i = 1; i <= MAX_SRC_CAP; ++i) {
  for (int l = 0; l <= i; l++) {
    pQueue q = initQueue(i);

    for (int j = 1; j <= l; ++j)
      insert(q, __llbmc_nondef_char());

    for (int k = 1; k <= MAX_DST_CAP; ++k) {
      pQueue q2 = initQueue(k);
      copyNoDuplicates(q2, q);
      finishQueue(q2);
    }
    finishQueue(q);
  }
}
```

Listing 7.9: Wrapper function to establish pre-state for checking contract of `copyNoDuplicates` with LLBMC

For our running example, without further knowledge about the properties of functions `copyNoDuplicates` or `insert`, this initialization function has to generate all possible combinations of source and destination queues for the LLBMC tool to perform exhaustive search for bugs up to a certain bound. The code in Listing 7.9 shows the implementation of this initialization function used in the subsequent experiments and is parametrized by the maximum capacity of generated source and destination queues (given by macros `MAX_SRC_CAP` and `MAX_DST_CAP`). The two outer for-loops in this code range over increasing capacities of the source queue and the number of items to be contained in this queue. The queue is populated by calling insert with the LLBMC specification function `__llbmc_nondef_char()` as parameter, which returns an arbitrary value of type char (resulting in an underspecified constant in the proof obligation given to the SMT solver). Reclaiming memory that has been allocated in calls to `initQueue` is done, after the queue is no longer used, by the method `finishQueue` (implementation not shown here).

That this function indeed produces a program state satisfying the preconditions of all called methods warrants another proof obligation – in this case, constructing the data structures by calling the appropriate initialization and insert methods, one can easily see that the resulting state is well-formed according to the precondition of `copyNoDuplicates`.

```
pQueue copyNoDuplicates(pQueue dest, pQueue source)
    _(ensures \forall uint i; i < \result->count ==>
        (\exists uint j; j < source->count
            && \result->items[i] == source->items[j])
     && (\forall uint j; j < \result->count ==>
            (\result)->items[i] == \result->items[j] ==> i == j))
    _(ensures (\forall uint i; i < source->count ==>
        \exists uint j; j < (\result)->count
            && source->items[i] == (\result)->items[j]))
```

Listing 7.10: Method contract of function `copyNoDuplicates` for LLBMC

Experiment 1. *Using the initialization function on the preceding page, LLBMC is run on the translated source code based on the implementation of Listing 7.5, together with the contract shown in Listing 7.10 – this contract is an extended version of the postconditions given for our sample VCC verification in Listing 7.7.*

The invocation of LLBMC uses the following parameters:

`llbmc --only-custom-assertions --max-loop-iterations=`k

where k is the smallest number of loop iterations sufficient to exhaustively check the specification and depends on the values for `MAX_SRC_CAP` and `MAX_DST_CAP` – in this case, as both values are identical, k = `MAX_DST_CAP` + 1. The option to check only custom assertions omits generation of proof obligations uncovering, e.g., array access out of bounds or references to unallocated memory. Run time results for the different values for `MAX_SRC_CAP` (and matching `MAX_DST_CAP`) are shown in Figure 7.11a and b.

As expected, LLBMC detects the violation of a custom assertion and provides the label of the basic block this assertion is contained in, as well as the enclosing function name:

```
Error synopsis:
===============

Assertion failed: Custom assertion (assert or __llbmc_assert) does ↙
    ↳ not hold.

Error location:
===============

Error occurs in basic block "bb73" of function "copyNoDuplicates".
```

To get a more precise feedback, the name of the basic block referenced in the LLBMC output has to be mapped back to the original annotated C code – for this, LLBMC provides an option to generate a concrete counterexample from the model of the SMT prover that demonstrates violation of the property in form of a complete execution trace up to the first offending assertion. However, this trace is given on the intermediate program representation, also, the size of the trace often gets large due to function inlining and loop unrolling (e.g., for the setup of Experiment 1 with a maximum of three loop iterations, the counterexample amounts to almost 50 kLOC).

The basic block referenced in the previous LLBMC output is the last block in the execution trace and consists of the following three instructions:

```
bb73:                                        ; executed
 %_forall_res_10.0 = phi i8
     [ 1, %bb73.loopexit ], [ 0, %bb70 ]    ; unknown
 %1062 = sext i8 %_forall_res_10.0 to i32   ; unknown
 call void @__llbmc_assert(i32 %1062)       ; ERROR DETECTED
```

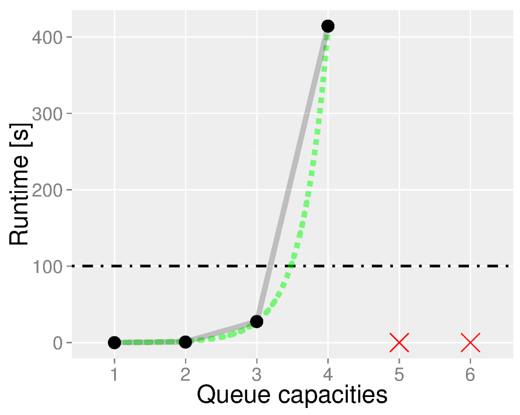
In this single static assignment representation, the failed assertion corresponds to the call to `__llbmc_assert`. This function invocation references variable `%_forall_res_10.0` (via variable `%1062`) – this variable name is introduced by our translation procedure from VCC annotations to C code. While inspecting this trace backwards from this basic block reveals the corresponding violated annotation in the original example, a simpler method would be to encode the location in the source code of the annotation in the variable name – this is a common technique used by other auto-active verification tools, e.g., Boogie.

In addition to the precise location of the violated annotation, the counterexample may be used to map the execution trace on the intermediate representation level back to the C code. As can be seen in the part of the counterexample above, the SMT solver does not assign values to all variables in the model, hence some variables in the execution trace are underspecified (marked as "unknown" above). Extracting the relevant information from this counterexample, in our case the values for the source and destination queue capacity, is non-trivial for the user without further tool support. Here, these values can be obtained by tracing calls to `initQueue` for both C queue structures used in the last invocation of `copyNoDuplicates` in the complete execution trace (not shown here).

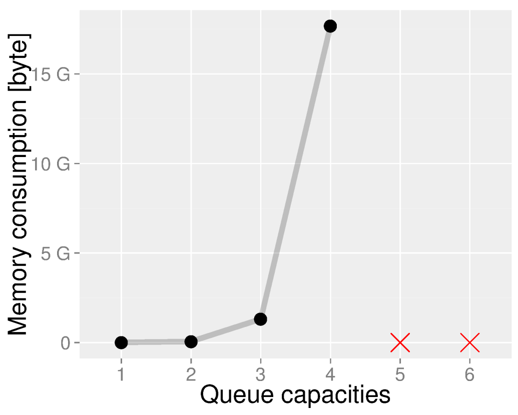
7.5.2. Improving Performance of Specification Checking

Unsurprisingly, this first experiment generating a large proof obligation for all possible inputs to function `copyNoDuplicates` does not scale well: Which path is taken for the loops in the implementation for functions `insert` and `copyNoDuplicates` in all cases either depends on the (symbolic) value of the elements in source or destination queue or the number of contained elements in the queue – this leads to case distinctions over the loop condition for each (nested) loop body execution.

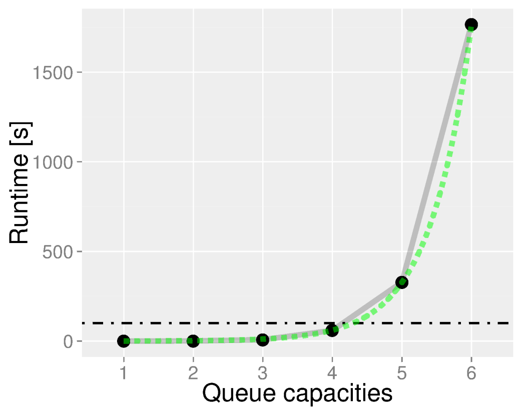
Curve fitting for the run time data points in Figure 7.11 using the R tool for statistical computing [RCT12] illustrates the exponential growth of run times, as well as peak memory consumption, in the size of queue capacities. With the given memory limit of 32 GiB, bounded checking succeeds for up queues up to size four.



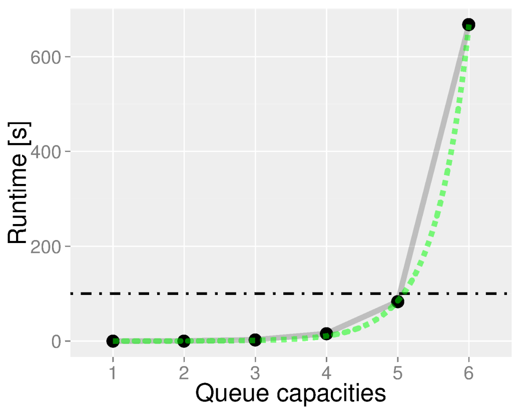
(a) Total run times for Experiment 1 on page 137, checking all combinations of queue capacities up to x in a single LLBMC invocation. Green dotted graph shows $f(x) = e^{-4.84+2.72x}$



(b) Peak memory consumption for experiments in (a).



(c) Sum of run times for checking each combination of queue capacities of (a) in separate LLBMC invocations. Green dotted graph shows $f(x) = e^{-2.64+1.69x}$



(d) Maximum of run times for LLBMC invocations in (c). Green dotted graph shows $f(x) = e^{-5.87+2.06x}$

Figure 7.11.: Performance of exhaustive specification checks with LLBMC. x-axis: maximum capacity of both source and destination queue (`MAX_SRC_CAP` and `MAX_DST_CAP`) in Listing 7.9 on page 136. Red crosses denote LLBMC runs with memory consumption exceeding limit of 32 GiB. Dotted horizontal line marks 100 s for comparison between graphs with different y-axis scaling. Green dotted graphs represent functions fitted to the data points.

Parallelizing Specification Checking To achieve better run times, one possibility is to split the verification task into smaller problems and to distribute the work onto several computing instances – a first candidate for this are the loops in the initialization function:

Experiment 2. *Given the setup as in Experiment 1, both loops ranging over the source and destination queue capacities in the main wrapper function (starting in line 1, respectively line 8 in Listing 7.9) are replaced each by a single assignment to variable i, respectively k. LLBMC is then run on the* `MAX_SRC_CAP` × `MAX_DST_CAP` *resulting C files, each corresponding to one iteration of the outer and inner for-loops of Experiment 1. Run time results of this experiment are shown in Figure 7.11c and d.*

Although the large amount of LLBMC invocations in this experiment incurs additional overhead (e.g., by repeatedly parsing the intermediate LLVM representation of the implementation), overall run-time is improved – e.g., for `MAX_SRC_CAP` = `MAX_DST_CAP` = 4 by a factor greater than seven (see Figure 7.11c). Of course, the exponential growth of the run time does not change by partitioning the proof obligations, but the performance improvements achieved by this optimization step allows us to check the contract of `copyNoDuplicates` for queues up to capacity six.

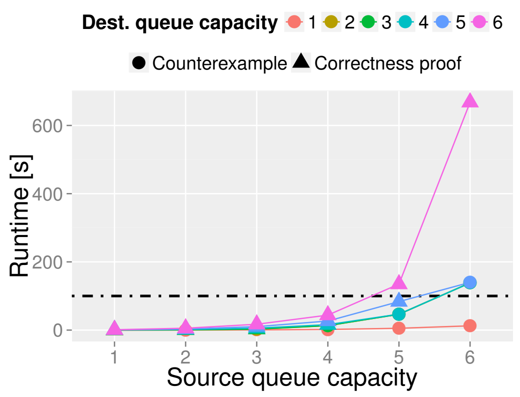
In addition to the improved totalized run time, distributing the different LLBMC invocations on multiple computing units in a cluster is another benefit of this approach. Figure 7.11d provides an estimation of the possible speedup by showing the maximum of run times of the different invocations. Compared to the sum of run times, distributing the workload improves time for feedback by LLBMC almost by a factor of three.

One reason for the improved performance is the number of loop unrollings needed in the two experiments. LLBMC uses a simple heuristic to determine a sufficient number of loop unrollings, however, due to the underspecified values for the count and capacity fields of the queues, which are used in the loop conditions, this heuristic is not applicable for our example. Instead, LLBMC uses the user-provided number of loop unrollings given as command line parameter.

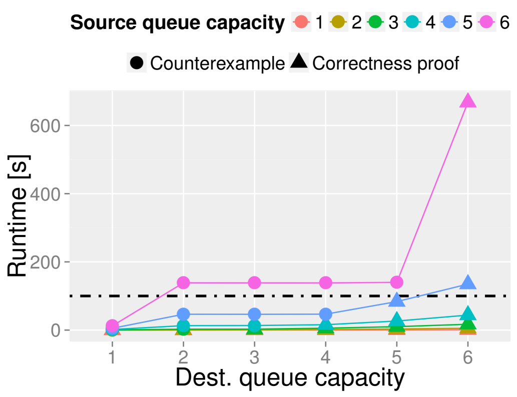
In addition to performance improvements, checking the properties of the implementation for single instances of different queue capacities allows us to analyze run time and memory consumption characteristics and their dependence on the size of the source and destination queue. Plots for run times and peak memory consumption as a function of source respectively destination queue capacity are shown in Figure 7.12.

Both run time and peak memory consumption of checking the single problem instances behave similarly when plotted against the capacity of source respectively destination queues. Firstly, we can identify an exponential growth of both numbers in the size of the source queue capacity. Secondly, considering the destination queue capacity in Figure 7.12b and (d), except the trivial case with a destination queue that can store only a single element, both run time and memory usage remain constant for all runs which produce a counterexample, i.e., whenever `dest.capacity < source.capacity`.

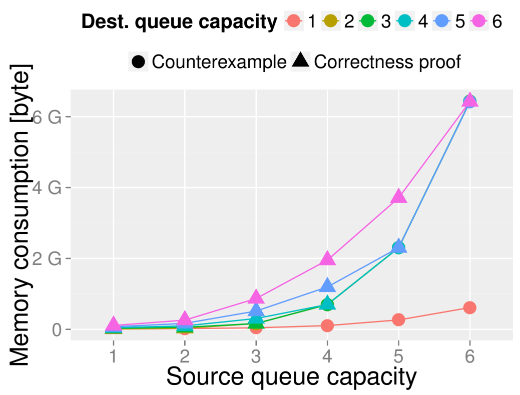
In all other cases, the program is correct w.r.t. its specification and *all* execution paths up to the given bound have to be explored by LLBMC, which explains the increase in run time for all instances where `dest.capacity` ≥ `source.capacity`.



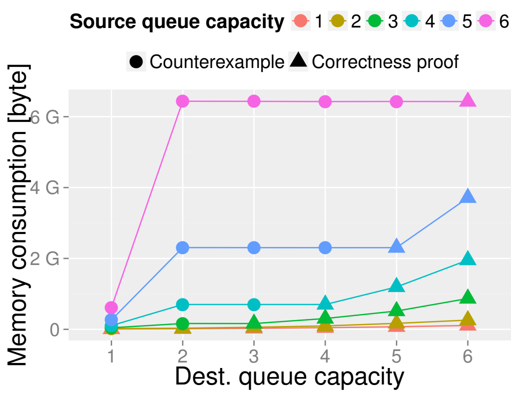
(a) Total LLBMC run time as a function of source queue capacity.



(b) Total run time as a function of destination queue capacity.



(c) Peak memory consumption as a function of source queue capacity.



(d) Peak memory consumption as a function of destination queue capacity.

Figure 7.12.: LLBMC performance results checking specifications for single combinations of source and destination queue capacities in Experiment 2 without ordering constraints of source queue elements. Dotted horizontal line marks 100 s for comparison between graphs with different y-axis scaling.

Beyond the point where the capacity of the destination queue matches the source queue, the run time and memory usage remains constant when further increasing destination queue capacity, provided the maximal number of loop iterations as passed to LLBMC is unchanged. This is due to the number of all loop iterations in the program execution (either directly or indirectly) depending only on the *source* queue capacity.

Still, for test cases with `dest.capacity` $\geq$ `source.capacity`, Figures 7.12b and (d) show an increase in run time and memory usage. This is due to the simple heuristic for the loop iteration parameter, which uses $\max($`dest.capacity`, `source.capacity`$) + 1$ as an estimation. For fixed source and destination queue capacities, run times and memory usage grow at least polynomially in the number of maximal loop unrollings (due to memory limitations not enough data points could be gathered to distinguish between polynomial and exponential growth of the functions).

Another possibility to split the specification checking task is to examine each postcondition on its own, instead of generating one proof obligation containing all postconditions as in all previous experiments. However, in our example for the implementation of `copyNoDuplicates`, run times for checking each postcondition with LLBMC do not differ significantly from performing the bounded correctness proof for all postconditions combined. Also, between the three different postconditions, the time needed for an exhaustive check was almost identical – this result agrees with the fact that all three postconditions have a similar structure of two nested quantifiers ranging over elements of the source respectively destination queue. Although splitting proof obligations did not improve run times in our example, in general, different complexity of the annotations in a specification to check may lead to earlier feedback in case of existing counterexamples.

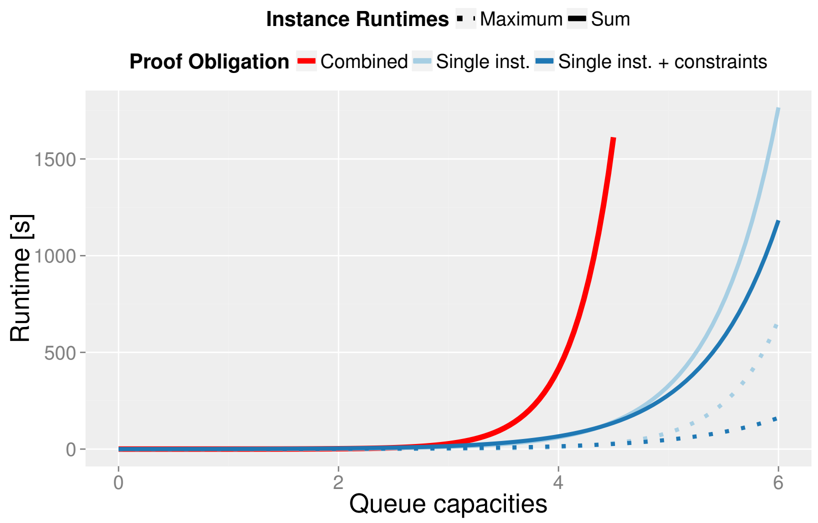


Figure 7.13.: Comparison of LLBMC performance for checking specifications using different optimization strategies

Using Domain Knowledge to Improve Performance One option to simplify proof search for the SMT solver is to reduce the search space by ruling out certain execution paths with the help of additional assumptions. In our example, we can make use of symmetries in the execution with the help of the fact that function `insert` stores queue elements in ascending order: any permutation of a sequence of elements inserted sequentially into an empty queue gives the identical resulting queue. The effect of using this information when checking specifications is examined in the following experiment:

Experiment 3. *Given the setup of Experiment 2 for checking single instances for different source and destination queue capacities, the wrapper function `main` is changed by adding a constraint in the for-loop inserting elements in the source queue (line 6). This constraint in the form of an LLBMC assumption states that an element inserted into the queue is smaller than or equal to the previously inserted element.*

While there is no appreciable change in memory consumption between LLBMC runs in Experiment 3 (with constraint) and Experiment 2 (without constraint), run times decrease significantly with the introduction of the assumption. A comparison between the run time performance results of the different experiments is shown in Figure 7.13.

7.6. Related Work

The insight that auto-active verification tools are reliant upon (and are lacking in) producing useful feedback for failed verification attempts is not new [LM10; PFW13]. To improve this situation, the output of verification tools like VCC, already pinpointing the exact annotation that could not be verified, is usually complemented with tools like the Boogie Verification Debugger (BVD) [GLM11], presenting counterexamples for the proof obligations on the level of the original program.

Still, due to the modular verification methodology of the deductive verification tools, these counterexamples are often not sufficient to retrace the concrete program execution leading to the violated specification (if there is any). This shortcoming has been identified and is addressed by many approaches, some of which we briefly discuss in the following.

One possibility to help the user in understanding failed verification attempts is to improve the presentation of counterexamples already given by the SMT back-ends of verification tools. For this, Müller and Ruskiewicz present a method that takes a program, together with a counterexample for a failed verification attempt and generates a program reproducing a concrete trace through the original code [MR11]. This program is annotated with debugging information that allows to use a debugger to trace through the original program while witnessing concrete states corresponding to the values of the counterexample. Execution of the generated program adheres to the modular verification semantics (e.g., calls to a function modify the state according to the postcondition and not via actual execution of the function body), allowing the user to identify mismatches between program and specification.

Closest to our idea of using bounded techniques to establish early and precise feedback in a new verification process is the work by Tschannen et al. [Tsc+13], published after our

FoVeOOS publication [Bec+12] this chapter is based on. They present *two-step verification* as the application of both modular deductive reasoning, as well as bounded techniques to identify the reason for verification failures, with a tool implemented as part of the verification framework AutoProof [Tsc+11] for the Eiffel programming language. In addition, implicit correctness properties of the program (e.g., valid memory accesses) are made explicit with so-called *implicit contracts* – in our case, both VCC, as well as LLBMC already include checks (respectively proof obligations) for most of these contracts. Using Boogie as back-end, which natively supports quantified formulas, removes the need to provide special handling of quantifiers compared to our approach. Although run-times also grow exponentially both in the number of loop unrollings and method inlining, compared to our experimental results, scalability of their technique is better.

Related to our technique of handling VCC specification constructs within the simpler LLBMC annotation language is the translation of specification for run-time checking, e.g., as described by Zee et al. [Zee+07] for checking specifications written for the verification tool Jahob [ZKR08] at run-time. Similar to our translation, bounded quantifiers are substituted by a finite iteration over all instances of the bound variable.

Considering the use of method inlining and unrolling loops, there are several alternatives: Firstly, the Boogie tool already provides a feature to verify loops and functions via inlining/unrolling instead of using contracts and loop invariants. Another option for bounded whole-program analysis on the Boogie language level is Corral [LQL12], improving the static inlining of Boogie by more efficient stratified inlining, together with variable abstraction: Search for counterexamples is first executed using approximations of method calls instead of inlining the corresponding method bodies. Only in case this analysis was inconclusive, the implementations of the methods are inlined.

Another technique that is applied to generate feedback for verification attempts is symbolic execution, e.g., as used by Boogaloo [PFW13] for the Boogie language, or an extension for the Pex tool presented by Vanoverberghe et al. [Van+08]. Both methods are able to provide concrete executions that demonstrate failed verification attempts (in addition, Boogaloo allows the user to inspect concrete program executions that are correct w.r.t. the specification). Program states are given internally in terms of symbolic values, together with path conditions (also called state constraints) – both tools differ, among others, in how quantified constraints are handled. A notable difference between Boogaloo and our technique is that for Boogaloo, method implementations are not required to perform analysis if a contract for the method exists.

The idea to use the complementary strengths of deductive program verification and model checking is not new – for a general overview of the possible combinations of both approaches, albeit for reactive systems, we refer to the work of Uribe [Uri00].

Understanding the feedback provided by program analysis tools is not only an issue with auto-active verification systems, but also for (bounded) model checkers. To support the user, the presentation of the counterexamples provided by the systems may be simplified and enriched by auxiliary information: one example is the *explain* tool [GKL04] for the bounded model checker CBMC. Based on an existing counterexample from CBMC, this tool generates further concrete execution traces that are either (a) also failing for the property in question, but are different to the original counterexample or (b) similar to the execution of the counterexample but which satisfy the property violated in the original program. Although not directly applicable for annotation-based deductive verification, these techniques may be used to improve feedback of our combination of VCC with LLBMC.

7.7. Conclusion and Future Work

Auto-active verification tools like VCC have changed the mode of interaction between the users and the prover compared to traditional interactive systems – for a typical verification task, the user can expect fast feedback, at the same time, he or she is reliant on these quick turnaround times due to the restricted way to interact with the tool.

In this chapter, we presented a method to enable earlier (and more precise) feedback for certain specifications and an improved verification process along with it, named CEGMAR. Based on the translation of VCC annotations to input of the LLBMC software bounded model checker, we implemented a prototype that is able to check specifications of annotated programs using a subset of VCC specification features. To facilitate integration of this validation technique into the verification process, we also developed extensions for an IDE (not presented here) that is able to represent feedback given by LLBMC to the user on the C implementation level.

Before this technique can be used productively for the verification of realistic software systems, the following three issues have to be solved in future work:

Improving Scalability Evaluation of the performance of our specification checking tool demonstrated feasibility of the approach – for a small (but non-trivial example) the tool is quickly able to provide counterexamples. At the same time, the experiments showed that scalability to larger execution traces and programs is a major problem for successful application of our method to real programs.

As a first step to mitigate this issue, we presented two possibilities to split the bounded verification task and distribute the resulting proof obligations to several CPUs or computers in a cluster. For our running example the run time of LLBMC grows exponentially in the length of the executed program trace to be analyzed. As a result, parallelization only allowed increasing the size of the involved queues, which are responsible for the length of the program trace, by two. We have not exhausted all opportunities to divide the verification task into smaller pieces: Further candidates are the conjunctions of a single postcondition that could be checked separately, also for bounded quantification, replacing the quantified formula by a conjunction of all possible instantiations of the bound variable results in a verification condition that can be distributed on several computing units.

Another option to improve on the time needed to get feedback from LLBMC is to reduce the size of the execution trace and the amount of properties to be checked. The usual approach for SBMC tools is to minimize the state space and the execution trace length by using primitive data types with a smaller range for variables in the implementation to be analyzed when possible (e.g., `char` instead of `int`), as well as choosing inputs for the program such that the length of the execution trace is short.

Success of this technique depends on whether the small model hypothesis is admissible, i.e., counterexample can be expected to be found even in short program executions.

The silver bullet to drastically reduce the length and number of execution traces to be analyzed is modularization – how to adapt SBMC tools to split a program into smaller models and check each individually is largely still an unsolved issue, especially if the premiss of using SBMC as a push-button technique is adhered to. For our purposes, we can loosen this restriction, as user interaction is already required to provide the annotations for deductive verification. An obvious way to modularize the bounded verification task is to provide additional wrapper functions that construct intermediate program states similar to the main function in Listing 7.9 on page 136.

For our purposes, we can also reduce the number of annotations that have to be checked, by making use of the distinction of auxiliary and requirement annotations – in a first step, testing only the requirement specification is sufficient. To distinguish the different annotations, the specification language has to provide a syntactical distinction between auxiliary and requirement annotations, as suggested in Chapter 5.

Concerning the main component used in our specification checking tool chain, the bounded model checker, different variants may be pursued in the future in increase efficiency: LLBMC uses a flat memory model to faithfully capture low-level C operations. The LLBMC tool authors mention that this memory model incurs a performance overhead compared to a typed memory model, but this difference in performance is usually offset by simplification steps in preprocessing [MFS12]. Given the reduced scope of simplification options when checking non-trivial specifications as in our running example (e.g., all program branches are relevant for the properties in question and cannot be pruned) this argument may no longer be valid and a typed model may improve performance for our use case.

LLBMC translates the bounded correctness property to a formula in quantifier free first-order logic (QF_AUFBV). However, program specifications for VCC often depend on quantification as seen in our running example. The translation of quantified formulas to loops necessary for property checking with LLBMC discards the additional information provided by the structure of the formula. Changing the bounded model checker to an alternative using an SMT solver back-end supporting quantifiers is part of future work – for this, we have started with first experiments with the bounded verification tool InspectJ [LNT12] for Java.

Improving the User Interface One motivation to use bounded model checking to complement deductive verification tools was to improve the feedback in case of a "spurious" counterexample presented by the verification system (e.g., in case auxiliary annotations are not sufficient to prove that the program is indeed correct w.r.t. the requirement specification). Concrete execution traces violating an assertion, as provided by SBMC, may help the user in understanding the cause of the specification violation. With the current output of LLBMC, relevant information has to be extracted manually.

Based on our experience with the VCC verification system, presenting feedback on the same level as the usual interaction between the user and the system (i.e., at the level annotations are given by the user: in the source code) helps to use the system efficiently by avoiding switching contexts. As a first feedback mechanism, we have adapted the existing presentation of errors in the program by “squiggly lines” to also notify the user of specification violations as a quick graphical display of the precise error location. One idea for future work is to also integrate display of information of the counterexample execution trace to be used in debugger of the IDE in the manner of the CounterExample Execution (CEE) tool [MR11].

Another point of user interaction is the necessity to provide a wrapper function for LLBMC that establishes a program state satisfying the precondition of the functions to test. Automatically generating an approximation of this function from the preconditions of the functions involved would enable a fully automatic specification checking process – this functionality is out of the scope of this thesis and part of future work.

8. Using Data Abstractions in Annotation-based Verification Systems

8.1. Specifying Operations on Abstract Data Types – A Simple Case Study

One desirable quality of any annotation-based program verification system is to provide a specification methodology that allows for concise and comprehensible annotations. Complexity of the verified program often correlates with the complexity or amount of annotations needed (e.g., especially as control of proof search has become a part of the annotations), aggravating the need for good specification languages.

One way to improve on existing specification constructs is to use specification patterns akin to software design patterns, capturing best practices and common idioms. Identifying these patterns may also give rise to new language keywords, allowing for more succinct annotations. An example of a VCC specification pattern already shown is to restrict updates to fields of C structures by introducing a volatile copy of the field, coupling both using an invariant and placing two-state invariants on the volatile copy.

Of course, different specification methodologies suggest different patterns and approaches – here we present examples for two methodologies by demonstrating for each verification tool implementations that solve a common problem, together with their functional specification. We then compare the two specification approaches and conclude with our proposal to use an established formalism to improve on data abstractions in annotation-based verification systems.

The examples we use as illustration are taken from the 1st Verified Software Competition [Kle+11]. The goal of the competition was to implement, formally specify and verify an algorithm that solves a problem defined in natural language. Our example is based upon the following requirement:

> **Problem: Searching a linked list.** Given a linked-list representation of a list of integers, find the index of the first element that is equal to zero. Show that the program returns a number i equal to the length of the list if there is no such element. Otherwise, the element at index i must be equal to zero, and all the preceding elements must be non-zero.

Several solutions for this problem were submitted within the competition using different verification tools and specification methodologies. Without the time limit of the competition as restriction, almost all contestants were able to provide solutions afterward. This allows for a comparison of different specification styles – below, we present two examples for a successfully verified implementation for the linked list problem. Besides the specification in VCC using the ownership-based memory model, we describe a solution made with the KeY tool, which employs dynamic frames to address framing.

8.1.1. The VCC Approach

The particular solution presented here, consisting of a C implementation and a specification in the VCC language, was developed after the competition by the team "VC Crushers" (see the paper by Klebanov et al. [Kle+11]).[1] It is an optimized version of their competition solution – the amount of auxiliary annotations is kept to a minimum sufficient to verify the requirement specification. Another VCC formalization of the list data structure that introduces a more general abstraction of the data type suited for a large range of applications is also made available by the team (not shown here for simplicity of the example).

The concrete C implementation of the list data structure and the C method `find` that is a solution to the above problem definition is shown in Listing 8.1. Annotations are given in VCC syntax and are enclosed in labeled frames throughout the code.

Linked List Data Structure

In the implementation of the linked list data structure (`struct List`), the field `data` contains the value stored in a node of the list, and the field `tail` points to the rest of the list. Note that each node of the list also stores the length of its tail (including the node itself). In this implementation, the end of a list is thus not indicated by a null pointer or a sentinel node but by the value of `length` being zero.

The semantics of the length field as well as an abstract representation of the list's contents are specified by the object invariants in the block labeled $\langle OI \rangle$: For each node, information about the elements of the sublist starting at that node is stored in the map `vals` in ghost state. The invariant I_1 defines the abstraction relation between the list and its abstraction `vals`. The abstraction from linked list to array is needed because the built-in data type array allows quantification and recursion over the elements of the list. A direct quantification over the elements is not possible in first-order logic because reachability is not first-order definable.

To be able to access all elements of the list (via the `data` field of the structure), each node is given exclusive ownership to the next node in the list (`mine`-annotation of invariant I_2). In addition, the invariant I_2 implies acyclicity of the list, as the length is bounded by zero, and is decreasing for each element that can be reached via the `tail` pointer.

Implementation of the `find` Algorithm

Using the C data structure `List`, the `find` method can be implemented by iterating over the list's elements via the tail pointer in a `for`-loop. Annotations are used within the method at three locations: for the method contract $\langle MC \rangle$, the loop invariant $\langle LI \rangle$ and as an auxiliary annotation inside the loop body $\langle AN \rangle$.

[1]For better readability, we have slightly modified the source code of the example.

```
typedef struct List {
  int data;
  struct List *tail;
  unsigned length;
  _(ghost int vals[unsigned])
  _(invariant vals == \lambda unsigned i;                //(I1)
      i < length ? (i == 0 ? data
                          : tail->vals[i - 1])
               :  0                                                ⟨OI⟩
  )
  _(invariant length == 0                                //(I2)
      || (\mine(tail) && length == tail->length + 1))
} List, *PList;

unsigned find(PList l)
  _(requires \wrapped(l))
  _(ensures \result <= l->length)
  _(ensures \result < l->length
              ==> l->vals[\result] == 0)                           ⟨MC⟩
  _(ensures \forall unsigned i; i < \result
              ==> l->vals[i] != 0)
{
  PList p;
  for (p = l; p->length != 0; p = p->tail)
    _(invariant p->length <= l->length)                  //(I3)
    _(invariant p \in \domain(l))                        //(I4)
    _(invariant p->vals == \lambda unsigned j;           //(I5)
        j < p->length
          ? l->vals[l->length - p->length + j]                     ⟨LI⟩
          : 0)
    _(invariant \forall unsigned j;
        j < l->length - p->length ==> l->vals[j] != 0)
  {
    _(assert \forall unsigned j; j < p->tail->length
          ==> p->tail->vals[j] == p->vals[j + 1])                  ⟨AN⟩
    if (p->data == 0) {
      break;
    }
  }
  return l->length - p->length;
}
```

Listing 8.1: Annotated C source code of `find`

The method contract in block $\langle MC \rangle$ specifies that the behavior of the method conforms to its specification in natural language. The pre- and postconditions $\langle MC \rangle$, together with the invariants $\langle OI \rangle$ of the list data structure, capture the intended semantics of `find`. $\langle MC \rangle$ and $\langle OI \rangle$ constitute the method's *requirement specification*.

To be able to verify the implementation of `find` to be correct w.r.t. to its requirement specification, a set of four essential auxiliary annotations has to be provided the in form of loop invariants (block $\langle LI \rangle$). The invariants I_3 and I_5 state that the abstraction of the "iterator" `p` is always a suffix of the abstraction of the input list `l`. As each access to a list element `p->data` inside the loop has to be shown to be a valid access, additional justification has to be provided to VCC to be able to prove this. This justification is given with the invariant I_4 – the property depends on the fact that `p` is also a sublist of `l` in the concrete representation.

Even in this simple example, a non-essential auxiliary annotation is needed at location $\langle AN \rangle$ to be able to show that the loop body preserves the third loop invariant. It asserts that the `vals` fields related to two adjacent nodes are abstractions of the appropriate sublists starting at those nodes.

In order to come up with the right auxiliary annotations, in this case, the user has to know about the inner workings of VCC and the strengths of the underlying SMT solver Z3, an issue explained in detail in Chapter 5.

```
/*@ public normal_behaviour
  @  ensures 0 <= \result;
  @  ensures (\result < vals.length && vals[\result] == 0)
  @          || \result == vals.length;
  @  ensures (\forall int x; 0 <= x && x < \result; vals[x] != 0);
  @*/
public int search() { ... }
```

Listing 8.2: JML contract of method `search`

8.1.2. The KeY Approach

The second implementation for the linked list search assignment we will present in the following is verified using the KeY tool and was submitted after the verification competition by the KeY team (consisting of Vladimir Klebanov, Mattias Ulbrich and Benjamin Weiß). Although the solution is written in Java with JML annotations compared to the C implementation of the VCC team, the assignment exercises only few programming and specification language features so that we are able to compare both solutions.

Unsurprisingly, for a simple assignment like searching a list, the Java implementation used as basis for the solution by the KeY team as given in Listing 8.3 is almost identical to the C variant shown before. Whereas in the VCC version the length of the list was stored as part of the data structure to be used as termination condition of the loop, the Java implementation uses the null reference to recognize the end of the list.

```
class Node {
  private int head;
  private Node next;

  public int search() {
      Node jj = this;
      int i = 0;

      while(jj != null && jj.head != 0) {
          jj = jj.next;
          i++;
      }
      return i;
  }
}
```

Listing 8.3: Java implementation of class `Node` with method `search`

Again, the specification of method `search` resembles its VCC counterpart, making use of an abstraction of the linked list data structure by field `vals`, as shown in Listing 8.2.

One notable difference lies within the invariants of the linked list data type. First, compared to VCC, the JML specification features the notion of visibility of invariants, allowing to designate which invariants constitute the interface for the data structure. As the public specification of `search` uses the field `vals` of class `Node`, in this example, the coupling invariant between `head`, `vals` and the list successor is also made public:

```
class Node {
  //@ public ghost \seq vals;
  /*@ public invariant
    @   1 <= vals.length && vals[0] == head
    @   && next == null ==> vals.length == 1
    @   && next != null ==> vals[1..vals.length-1] == next.vals;
    @*/

  //@ public ghost \locset repr;
  /*@ public invariant \subset(this.*, repr) &&
    @   next != null ==> (\subset(next.*, repr)
    @                   && \subset(next.repr, repr)
    @                   && \disjoint(this.*, next.repr));
    @*/

  //@ public invariant next.\inv
  //@ accessible \inv: repr \measured_by vals.length;
```

The definition of the set of memory locations `repr`, starting in line 9, is used to specify non-aliasing properties and for the accessible clause in line 17. This clause gives the set of memory locations the invariants of class `Node` at most may depend on. Interestingly, the specification concerning field `repr` in this case are similar to the built-in ownership model of VCC, together with the annotation `\mine(tail)` in the C `List` definition: implicitly, the List structure "owns" its fields of primitive type (i.e., length and data), specified in JML with `\subset(this.*, repr)`. If the next list element exists, it is also part of the representation of the list (`\subset(next.*, repr)` respectively `\mine(tail)`); also, everything "owned" by the next list element is transitively owned by the current node (given by `\subset(next.repr, repr)` in case of JML and implicitly for VCC). Lastly, disjointness of the different list elements is specified by `\disjoint(this.*, next.repr))` as opposed to the same fact derivable from definition of ownership in VCC.

This example gives a first insight into the flexibility of the dynamic frames approach – at the same time, also the verbosity of this methodology compared to the fixed ownership system of VCC is apparent. Whether this trade-off between flexibility and conciseness is well-chosen also depends on the specific nature of the use cases of the verification tool. In the following, we will present a third variant of a possible specification approach for the running example, based on abstract data types.

8.2. Separation of Concerns: Annotation-based Verification and Algebraic Specifications

In the examples shown in the previous sections, annotations with different purposes are intermingled. In the following, we suggest to use techniques known from abstract data type specifications to provide a clean separation of requirement and auxiliary annotations, and to make the specification more readable.

Assuming that we have defined an abstract data type `IntList`, an abstraction function `abs` from concrete linked lists to abstract lists, as well as an abstract (specification) function `absfind` on `IntList`, a good method contract for `find` could look like this:

```
unsigned find(PList l)
  _(requires \wrapped(l))                           } ⟨MC⟩
  _(ensures \result == absfind(abs(l)))
```

This contract is very compact and easy to understand. It simply states that `find` returns the same integer value that is the result of the abstract operation `absfind` on the abstraction of the input list.

In the VCC example presented in Section 8.1.1, the equivalent of `absfind` is implicitly given by postconditions $\langle MC\rangle$ of method `find`. The abstraction function `abs` is concealed within the definition of structure `List`, namely in the invariants in $\langle OI\rangle$. As both the VCC and KeY version of the specification are similar to each other, the same is true for the JML specification of the Java implementation.

Of course, to complete the specification, we now have to define `IntList` and `absfind`. The required syntax is not available in VCC (yet). We suggest to use a syntax for abstract data type definitions based on the Common Algebraic Specification Language (CASL) [Mos04]. A possible definition then would look like this:

```
spec IntList =
  free type List ::= nil | cons(Int; List)
then vars i, e: Int; l, l': List
  op absfind : List -> Int
    * absfind(nil) = 0
    * absfind(cons(e, l)) = 1 when e = 0
                            else absfind(l) + 1
  ops append : List x List -> List
      tail : List -> List
    * append(nil, l) = l
    * append(cons(e,l'), l) = cons(e, append(l',l))
    * tail(nil) = nil
    * tail(cons(e,l)) = l
  within implementation find
end
```

To be able to specify the implementation of `find` with the help of the abstract data type, the data type is equipped with the externally visible operation `absfind` that captures the semantics of `find` according to the problem definition. In addition, the two operations `append` and `tail` are defined with the usual semantics.

Compared to the rudimental abstraction given by the map `vals` in Section 8.1.1, this specification is additional overhead in terms of lines of code. However, the compactness of the map specification is partly due to the fact that maps are a built-in feature of the verification tool. Furthermore, the flexibility offered by defining arbitrary abstract data types in our opinion clearly outweighs the annotation overhead, as we can choose the abstract data type representation that matches the implementation data types best. Lastly, the above definition is to a large extent reusable and can be seen as a sort of library definition.

To relate the concrete implementation data type `List` to its abstract data type counterpart `IntList`, we have two options: (A) defining an abstraction function `abs` from `List` to `IntList` (as already mentioned above), and (B) axiomatizing the abstraction using concrete implementations for the (abstract) constructors `nil` and `cons`.

For option (A), the abstraction function `abs` is defined in terms of the concrete implementation details (i.e., field `data` and pointer `tail`):

```
_(spec IntList abs(PList l)
   returns(l->length == 0 ? nil :                      }
                cons(l->data, abs(l->tail)))            } ⟨OI⟩
)                                                       }
```

With this definition, one of our goals is already achieved, namely providing a clearly differentiated and discernible specification construct that couples the concrete and abstract data types. However, this definition does not hide the implementation details of the linked list data structure from callers of the `find` method.

The alternative solution (B) uses concrete implementations of the constructors of the list data structures (not shown here). Then, no explicit definition of the abstraction function as in (A) is needed. The relation between the abstract and the concrete constructors can, for example, be specified as follows:

```
PList cons(int e, PList l)                                    }
  _(requires \wrapped(l))                                     } ⟨OI⟩
  _(ensures abs(\result) == cons(e, abs(l)) )                 }
```

Note that the above method contract of `cons` does not use any implementation details of the linked list data type in C, so that these details do not become part of the requirement specification.

Regardless of which alternative (A) or (B) is chosen, due to the separation of abstract and concrete representation of the list data type, the annotation overhead of the `List` data structure can be reduced, as demonstrated in the following structure definition.

```
typedef struct List {
  int data;
  struct List *tail;
  unsigned length;
  _(invariant \exists IntList l; abs(this) == l
       && (abs(this) == nil || \mine(tail)))
} List, *PList;
```

Only one invariant of `list` remains that is concerned with the state of the structure according to the VCC methodology (keyword `mine`), as well as enforcing the existence of an abstract element corresponding to the concrete instance of the list (ruling out cyclic lists).

Using the two "library" functions `tail` and `append`, almost all auxiliary annotations of our example shown in Listing 8.1 can be simplified – for the loop invariants at location $\langle LI \rangle$ the new annotations are:

```
    _(invariant p \in \domain(l))
    _(invariant \exists IntList front;
        append(front, abs(p)) == abs(l)
        && absfind(f) == 0)
```

Furthermore, the single non-essential annotation at location $\langle LI \rangle$ becomes:

```
    _(assert abs(p->tail) == tail(abs(p)))
```

8.3. Related and Future Work

The idea to use algebraic specifications for abstract data types for auto-active verification is not new – at the time our paper this chapter is based on was published, some systems already made use of some features based on algebraic specification, e.g., the Why software verification platform [FM07] or Verifast [JP08]. Since then, these existing features have been extended, respectively found their way into many other auto-active verification tools.

Adoption of this specification style also depends on the ability to verify properties relevant in practice and thus on the back-end prover used – in this regard, verification tools including interactive proof assistants (traditionally with good support for induction) have an advantage over tools only relying on SMT solvers (having an induction proof method is not the normal case, e.g., the commonly used SMT solver Z3 lacks a custom procedure for induction proofs).

In the following, we give a few relevant examples for the support of abstract specification of data types in various verification tools. These existing approaches to lay down a specific syntax for specifying abstract data types with their accompanying semantics notwithstanding, we claim that the developers should give consideration to the CASL framework, providing a solid foundation for more advanced specification features.

One example for existing ADT specification features can be found in the ANSI C specification language (ACSL), which included (polymorphic) logic types at least since 2008 as of version 1.2 [Bau+08]. Logic types may be given a definition, resulting in concrete logic types – functions on the elements of the type may then use pattern matching to deconstruct a compound value according to the constructors given in the definition. However, concrete logic types as specification language feature are marked as "experimental", i.e., "their syntax and semantics is not already fixed" up to the current version of ACSL (1.8) [Bau+14]. Concerning the adoption of ACSL in the implementation of verification tools, concrete logic types in Frama-C are incomplete in the most recent version of the verification tool (Neon, as of 2014).

Another system that supports specification of abstract data types is VeriFast – since the first mention of a prototypical implementation of the tool in 2008 [JP08], the tool allows for inductive (parametric) data type definitions, as well as their use in specification functions and in defining fixpoint functions through case distinctions along the constructors of a type. Although the specification language is based on separation logic, the approach to handle abstract data types with an SMT solver as backend, as described by Jacobs and Piessens [JP08], is not specific to this framework and should be adaptable for other auto-active tools like VCC.

Also, the VCC tool gained support for first specification primitives related to specification of abstract data types, introduced as of 2011.[2] A description of these features is given in the VCC tutorial [Coh+12] and manual [VCC12]. More advanced specification features, e.g., support for type refinement, are not yet included in the specification language.

[2]The corresponding change set in the source code repository of VCC is available at: https://vcc.codeplex.com/SourceControl/changeset/fb74c293e2ac

```
_(def \integer absfind(IntList l)
  {
    switch(l) {
        case nil(): return 0;
        case cons(e, tail):
            return (e == 0) ? 1 : absfind(tail) + 1;
    }
  }
)
```

Listing 8.4: Switch statement for inductive data types in VCC

We will demonstrate the concrete syntax of (parts of) the specification features for inductive types at our running example in this chapter, to compare this to the CASL-inspired specification as given in Section 8.2.

Inductive definitions of the abstract type `IntList` in VCC are stated with the `datatype` keyword, listing the different constructors of the type (using keyword `case`):

```
_(datatype IntList {
    case nil();
    case cons(\integer, IntList);
})
```

VCC supports mutually recursive type definitions (not shown here), but does not allow for parametric types – in our running example, we only used the concrete type of integer list, so this restriction of VCC is of no concern. The definition of specification function `absfind` closely resembles the CASL version, albeit with custom VCC syntax to state the return values of the function according to the structure of parameter `IntList l`, as shown in Listing 8.4.

For an application of these VCC specification language features in a larger context, see, e.g., the work on verification of cryptographic C programs by Dupressoir [Dup13].

8.4. Conclusion

In this chapter we have demonstrated the benefit of abstract, algebraic specifications for annotation-based specification languages – for a small but realistic example, we started by presenting the annotations necessary to allow for functional verification of the implementation using the default specification strategy and patterns of the corresponding verification systems. The amount of annotations needed for this simple implementation exemplifies the issues when specifying data structures at a low level of abstraction.

We argue that the specification language should lend itself to be able to express and reason about these simple heap data structures, like linked lists or trees, in a concise and elegant fashion – exposing an interface that is straight-forward to use for clients working with the corresponding data structures. While some pre-defined and fixed abstract data types can be found in virtually all software verification systems (e.g., sequences in KeY or maps in VCC) that allow for a more abstract specification, we have seen at the running example that this is not enough.

In contrast, we have shown for the simple running example how appropriately chosen (user-defined) abstract data types yield more concise annotations, compared to the usual specification overhead incurred. For this, we used the existing work on the Common Algebraic Specification Language (CASL) to provide the definitions of the abstract data type used in our case study, together with the operations on the data structure.

Besides concise contracts, the annotation language should expose the structure of the specification and separate different concerns: in our case there should be a clear distinction between well-formedness invariants of the data structure on the concrete implementation level, coupling invariants between the abstract and concrete specification layers, as well as annotations describing properties of the abstract data type. Currently, in VCC, annotations serving different purposes are all indistinguishable from one another.

We have only used a small fraction of the feature set provided by CASL – in particular, to be able to specify larger, more complex software systems, two important questions have to be addressed: (a) How to construct compound specifications from more elementary building blocks, and (b) how to bridge the gap between the abstract specification and the concrete implementation.

For the former issue, CASL introduces *architectural* specifications [BST99], together with mechanisms to compose, extend or modify specifications by means of *structural constructs* (such as hiding components of a specification or extending specifications by additional axioms describing properties of the type defined). The latter problem may be handled with step-wise refinement of specifications until a specification is obtained that is concrete enough such that the programmer can easily write down an implementation that itself refines the abstract system specification – for an example of specification refinement in CASL, we refer to the work by Bidoit et al. [BST99].

In this chapter, we have only looked at how to specify using primitives from CASL, but excluded *reasoning* about correctness of the implementation w.r.t. this specification. Normally, in logical frameworks with interactive provers (like Isabelle/HOL) where specification of data types using inductive definitions is common, verification is performed with the help of structural induction. With auto-active systems, the SMT solvers used as back-ends often do not provide explicit tactics to conduct induction proofs and proof obligations have to be sufficiently encoded, e.g., as shown by the work of Leino [Lei12], enabling automation of inductive reasoning with SMT solvers [Lei13].

9. Conclusions

Current verification methodologies and tools have come far since the emergence of deductive software verification in the late 1960s – they allow the user to verify full functional properties of programs on source code level for programming languages used in practice and can be applied to complex software of substantial size – a good example for achievable results are the L4.verified project and both parts of Verisoft. The effort needed in specification and verification of such software systems, however, restricts the application of deductive verification to selected safety- or security-critical parts of the system – to be successfully applied in an industrial setting, formal verification techniques have to stand up against established quality assurance measures of software engineering. One aspect of this thesis is, on the one hand, to demonstrate feasibility of microkernel verification, but also to pinpoint obstacles for introducing formal software verification in the software engineering process.

For this, in the first part of this thesis, we presented the specification strategy for the verification of the correctness of the preemptible microkernel PikeOS used for system virtualization. The foundational specification methodology we used was largely developed as part of the Hypervisor sub-project within Verisoft XT and has shown to be flexible enough to accommodate many concurrent system setups – one example being PikeOS.

Compared to verification of the academic system in the first phase of Verisoft, where Isabelle/HOL was applied, for the verification of PikeOS (and Hyper-V) in Verisoft XT, the VCC tool was used. VCC was developed as part of Verisoft XT with the goal of providing a high degree of automation in verifying functional properties of concurrent C programs – interaction with VCC takes place at the source code level, with the user providing specifications in form of program annotations, an interaction paradigm called "auto-active".

Correct operation of PikeOS was specified in terms of an abstract system with the help of VCC's specification state, the valid transitions of this abstract kernel given by two-state invariants. Adherence of the PikeOS implementation to this specification was then shown using a forward simulation proof with the VCC tool. The abstract state that corresponds to the concrete state after a sequence of steps of the concrete kernel is in the VCC methodology given explicitly by the verification engineer with the help of explicit updates to the specification state. Both abstract and concrete state are coupled through invariants; making correspondence of the layers another proof obligation to be discharged.

If nothing else, the bottom line of this thesis and our primary conclusion from the Verisoft XT project is that current auto-active verification systems are powerful enough to be successfully applied to complex, real-world concurrent systems. Still, verification of a software system of moderate size like PikeOS takes considerable effort – also reflected by the fact that the verification of the whole implementation of PikeOS could not be completed within the time frame of the Verisoft XT project.

It is only due to the significant progress of deductive program verification tools in recent years that application of full functional verification to complex concurrent software has become a conceivable task. However, to be able to efficiently verify large software, further improvements are necessary. As the modules that can be verified using current tools get larger and more complicated, it becomes apparent that the process of finding the right specification is now the bottleneck.

We have identified challenges for program-level software verification that arose during the verification of PikeOS within the scope of the Verisoft XT project and that contribute to the problem of coming up with sufficient code annotations.

In the second part of this thesis, we have examined consequences of the auto-active verification paradigm of VCC and presented techniques to advance the state of the art of deductive program-level verification by addressing the shortcomings identified as part of the PikeOS case study.

One implication of the currently implemented design decisions of auto-active verification tools like VCC is that, besides the requirement specification, there is a certain class of annotations that is indispensable for proof construction, as explained in Chapter 5. The problems with annotation-based verification arise as the lines between the requirement specification, auxiliary information needed for proof construction, and information for proof-search guidance get blurred. As a consequence, users are often surprised that they cannot omit certain non-requirement annotations. The need to understand both the inner workings of the verification tool and the different classes of annotations contradict the design goal of auto-active systems to use the tool as a "black box". To give a precise description of the different kinds of annotations, we introduce the term of *annotation completeness* for verification systems and the distinction of auxiliary annotations between *essential* and *non-essential* annotations in Chapter 5.

One of the most important aspects of program verification systems for the application to safety- or security-critical software is soundness, which depends for a typical deductive program verification tool both on an implementation part and a set of calculus rules or an axiomatization. To improve trust in the verification system, we have demonstrated how the software testing approach may be used as a light-weight method to find errors in the axiomatization (or calculus rules) of a verification tool in Chapter 6. For this, we have introduced a coverage metric for testing verification tools that describes to which extent a test case exercises the axiomatization of the tool under test. We have also provided first evidence for the utility of the coverage metric by analyzing the axiomatization coverage in two case studies (the KeY system and the VCC tool) and demonstrated that increasing this coverage allows one to detect bugs in the axiomatization, respectively rule base.

In general, in order to handle verification of large software systems efficiently, we claim that better support from the verification tool is needed to get from verification of individual functions to verification of whole software systems. This would have to include support in finding the right modularization and abstraction. A first step in this direction is to provide the user with feedback in case of interdependent function specifications, so that mismatching contracts are discovered early in the specification process (see Chapter 7).

Besides annotation-based specifications, which are well suited for describing implementations at source-code level, there is a need for further specification constructs tailored to describe the system properties at higher levels of abstraction. Amongst others, these formalisms should allow to integrate knowledge of system developers even before starting the specification process at code level. To improve the annotation-based specification language of VCC, we propose to make use of established formalisms for abstract data types like CASL, to be able to compactly specify common implementation data structures (Chapter 8).

Future work includes to complete the prototypical implementation of the combination of software bounded model checking and deductive verification to provide the user with early feedback as presented in Chapter 7. Using this implementation, the next step would be an evaluation whether our technique improves the user's efficiency in verifying complex software systems, compared to the current auto-active verification approach.

For the axiomatization coverage metric introduced in Chapter 6, further case studies are needed to determine whether the coverage definition is a sensible testing metric, e.g., by writing additional test cases targeted at covering selected axioms that are not yet covered by existing tests for the tool under test. In addition, a more fine-grained coverage metric has to be developed that takes the syntactical structure of the axioms into account.

Based on our suggestion to re-use specification constructs for abstract data types introduced by the Common Algebraic Specification Language (Chapter 8), the details for an actual extension of the VCC specification language have obviously to be worked out – also, the specification constructs for abstract data types that have been implemented by the VCC developers in the meantime have to be taken into account.

Another line of research we consider promising is the combination of auto-active and more traditional, interactive verification approaches. Some challenges presented in the first part of this thesis are to a certain degree a consequence of the auto-active verification approach and the design decisions made in VCC, like the inflexible module granularity or the amount of annotations needed. Other issues mentioned affect both auto-active tools and interactive verification systems: entangled module specifications are of concern to all modular verification methodologies, also change management is a major issue in verifying evolving, real-world software.

For some of these issues shared by both verification approaches alike, interactive proof construction offers significant advantages – amongst others, guiding the prover in finding a proof (similar to finding sufficient annotations in case of VCC) is supported by being able to inspect the current proof state. Interactive proof systems like Isabelle/HOL also are more flexible concerning specification formalisms.

When a successful formal verification of complex and large software systems at code level is desired, we believe a combination of both interactive and auto-active specification and verification approaches could be helpful – similar to verification of certifying algorithms using VCC, together with Isabelle/HOL as shown by Alkassar et al. [Alk+11]. Using this combination, auto-active tools would allow for efficient verification of source-code level properties closer to the hardware, the resulting functional properties of the code abstracting from implementation details – whereas interactive verification systems may be used to verify complex system properties on a suitable, user-defined abstraction of the system.

Bibliography

[AA06] K. Adams and O. Agesen. “A comparison of software and hardware techniques for x86 virtualization”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*. (San Jose, USA, Oct. 21–25, 2006). Ed. by J. P. Shen and M. Martonosi. ACM, 2006, pp. 2–13. DOI: 10.1145/1168857.1168860.

[Abr96] J. R. Abrial. *The B-Book: Assigning programs to meanings*. New York, USA: Cambridge University Press, 1996. ISBN: 0-521-49619-5.

[AD10] K. Y. Ahn and E. Denney. “Testing first-order logic axioms in program verification”. In: *Tests and Proofs, 4th International Conference, (TAP 2010)*. (Málaga, Spain, July 1–2, 2010). Ed. by G. Fraser and A. Gargantini. Vol. 6143. LNCS. Springer, 2010, pp. 22–37. DOI: 10.1007/978-3-642-13977-2_4.

[AL91] M. Abadi and L. Lamport. “The existence of refinement mappings”. In: *Theoretical Computer Science* 82.2 (1991), pp. 253–284. DOI: 10.1016/0304-3975(91)90224-P.

[Alk+10a] E. Alkassar, E. Cohen, M. A. Hillebrand, M. Kovalev, and W. Paul. “Verifying shadow page table algorithms”. In: *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*. (Lugano, Switzerland, Oct. 20–23, 2010). Ed. by R. Bloem and N. Sharygina. IEEE, 2010, pp. 167–174.

[Alk+10b] E. Alkassar, E. Cohen, M. A. Hillebrand, and H. Pentchev. “Modular specification and verification of interprocess communication”. In: *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*. (Lugano, Switzerland, Oct. 20–23, 2010). Ed. by R. Bloem and N. Sharygina. IEEE, 2010, pp. 167–174.

[Alk+10c] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. “Automated verification of a small hypervisor”. In: *3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010)*. (Edinburgh, UK, Aug. 16–19, 2010). Ed. by G. T. Leavens, P. W. O’Hearn, and S. K. Rajamani. Vol. 6217. LNCS. Springer, 2010, pp. 40–54. DOI: 10.1007/978-3-642-15057-9_3.

[Alk+10d] E. Alkassar, W. J. Paul, A. Starostin, and A. Tsyban. “Pervasive verification of an OS microkernel – Inline assembly, memory consumption, concurrent devices”. In: *3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010)*. (Edinburgh, UK, Aug. 16–19, 2010). Ed. by G. T. Leavens, P. W. O’Hearn, and S. K. Rajamani. Vol. 6217. LNCS. Springer, 2010, pp. 71–85. DOI: 10.1007/978-3-642-15057-9_5.

[Alk+11] E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. "Verification of certifying computations". In: *23rd International Conference on Computer Aided Verification (CAV 2011)*. (Snowbird, USA, July 14–20, 2011). Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 67–82. DOI: 10.1007/978-3-642-22110-1_7.

[Alk+12] E. Alkassar, E. Cohen, M. Kovalev, and W. J. Paul. "Verification of TLB virtualization implemented in C". In: *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012)*. (Philadelphia, USA, Jan. 28–29, 2012). Ed. by R. Joshi, P. Müller, and A. Podelski. Vol. 7152. LNCS. Springer, 2012, pp. 209–224. ISBN: 978-3-642-27704-7. DOI: 10.1007/978-3-642-27705-4_17.

[ARS05] W. Ahrendt, A. Roth, and R. Sasse. "Automatic validation of transformation rules for Java verification against a rewriting semantics". In: *12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*. (Montego Bay, Jamaica, Dec. 2–6, 2005). Ed. by G. Sutcliffe and A. Voronkov. Vol. 3835. Springer. 2005, pp. 412–426. DOI: 10.1007/11591191_29.

[Ast+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. "CASL: the common algebraic specification language". In: *Theoretical Computer Science* 286.2 (2002). Current trends in Algebraic Development Techniques, pp. 153–196. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(01)00368-1.

[Bar+] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. At http://www.smt-lib.org/.

[Bar+06] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. *MOBIUS: Mobility, Ubiquity, Security*. Ed. by U. Montanari, D. Sannella, and R. Bruni. Vol. 4661. LNCS. Springer, 2006, pp. 10–29. DOI: 10.1007/978-3-540-75336-0_2.

[Bau+08] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language*. Tech. rep. (Version 1.2). Mar. 2008.

[Bau+09a] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. "Better avionics software reliability by code verification". In: *embedded world Conference*. (Noremberg, Germany). Ed. by M. Sturm. WEKA Fachmedien GmbH, Mar. 2009. ISBN: 978-3-7723-3798-7. URL: http://formal.iti.kit.edu/~bormer/pub/embeddedWorld2009.pdf.

[Bau+09b] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. "Formal verification of a microkernel used in dependable software systems". In: *28th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2009)*. (Hamburg, Germany, Sept. 15–18, 2009). Ed. by B. Buth, G. Rabe, and T. Seyfarth. Vol. 5775. LNCS. Springer, Nov. 2009, pp. 187–200. DOI: 10.1007/978-3-642-04468-7_16.

[Bau+10] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. “Ingredients of operating system correctness”. In: *embedded world Conference*. (Nuremberg, Germany). Ed. by M. Sturm. WEKA Fachmedien GmbH, Mar. 2010. ISBN: 978-3-7723-1012-6. URL: http://formal.iti.kit.edu/~bormer/pub/embeddedWorld2010.pdf.

[Bau+11] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev. “Proving memory separation in a microkernel by code level verification”. In: *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW 2011)*. (Newport Beach, California, Mar. 28–31, 2011). Mar. 2011, pp. 25–32. DOI: 10.1109/ISORCW.2011.14.

[Bau+12] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. “Lessons learned from microkernel verification: Specification is the new bottleneck”. In: *Proceedings of the Seventh Conference on Systems Software Verification (SSV 2012)*. (Sydney, Australia, Nov. 28–30, 2012). Ed. by F. Cassez, R. Huuck, G. Klein, and B. Schlich. Electronic Proceedings in Theoretical Computer Science 102. 2012. DOI: 10.4204/EPTCS.102.4.

[Bau+14] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. Tech. rep. Version 1.8 – Neon-20140301. 2014.

[Bau14] C. Baumann. “Ownership-Based Order Reduction and Simulation in Shared-Memory Concurrent Computer Systems”. PhD thesis. Saarland University, Saarbrücken, 2014. URL: http://www-wjp.cs.uni-saarland.de/publikationen/Ba14.pdf.

[BB09] C. Baumann and T. Bormer. “Verifying the PikeOS microkernel: First results in the Verisoft XT avionics project”. In: *Doctoral Symposium on Systems Software Verification (DS SSV 2009)*. (Aachen, Germany, June 22–24, 2009). Ed. by R. Huuck, G. Klein, and B. Schlich. Aachener Informatik-Berichte AIB-2009-14. RWTH Aachen University, June 2009. URL: http://aib.informatik.rwth-aachen.de/2009/2009-14.pdf.

[BB12] B. Beckert and T. Bormer. “Lessons learned from microkernel verification”. In: *Pre-Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)*. (Bamberg, Germany). Ed. by G. Lüttgen and S. Merz. 2012.

[BBK12] B. Beckert, T. Bormer, and V. Klebanov. “Improving the usability of specification languages and methods for annotation-based verification”. In: *9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*. (Graz, Austria, Nov. 29–Dec. 1, 2010). Ed. by B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue. Vol. 6957. LNCS. Springer, 2012, pp. 61–79. ISBN: 978-3-642-25270-9. DOI: 10.1007/978-3-642-25271-6_4.

[BBW13a] B. Beckert, T. Bormer, and M. Wagner. "A metric for testing program verification systems". In: *Seventh International Conference on Tests and Proofs (TAP 2013)*. (Budapest, Hungary, June 16–20, 2013). Ed. by M. Veanes and L. Viganò. Vol. 7942. LNCS. Springer, 2013, pp. 56–75. DOI: 10.1007/978-3-642-38916-0_4.

[BBW13b] B. Beckert, T. Bormer, and M. Wagner. "Heuristically creating test cases for program verification systems". In: *10th Metaheuristics International Conference (MIC 2013)*. (Singapore, Aug. 5–8, 2013). 2013.

[Bec+12] B. Beckert, T. Bormer, F. Merz, and C. Sinz. "Integration of bounded model checking and deductive verification". In: *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2011), Revised Selected Papers*. (Turin, Italy, Oct. 5–7, 2011). Ed. by B. Beckert, F. Damiani, and D. Gurov. Vol. 7421. LNCS. Springer, 2012, pp. 86–104. DOI: 10.1007/978-3-642-31762-0_7.

[Bec+14a] B. Beckert, T. Bormer, R. Goré, M. Kirsten, and T. Meumann. "Reasoning about vote counting schemes using light-weight and heavy-weight methods". In: *8th International Verification Workshop (VERIFY 2014)*. (Vienna, Austria, July 23–24, 2014). 2014.

[Bec+14b] B. Beckert, R. Goré, C. Schürmann, T. Bormer, and J. Wang. "Verifying voting schemes". In: *Journal of Information Security and Applications* 19.2 (2014), pp. 115–129. DOI: 10.1016/j.jisa.2014.04.005.

[Bel05] F. Bellard. "QEMU, a fast and portable dynamic translator". In: *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*. (Anaheim, USA, Apr. 10–15, 2005). USENIX, 2005, pp. 41–46. URL: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html.

[Bev89] W. R. Bevier. "Kit: A study in operating system verification". In: *IEEE Transactions on Software Engineering* 15.11 (1989), pp. 1382–1396. ISSN: 0098-5589. DOI: 10.1109/32.41331.

[Bey+06] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. "Putting it all together – Formal verification of the VAMP". In: *International Journal on Software Tools for Technology Transfer* 8.4-5 (2006), pp. 411–430. ISSN: 1433-2779. DOI: 10.1007/s10009-006-0204-6.

[Bha+12] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. "Graph-based analysis and prediction for software evolution". In: *34th International Conference on Software Engineering (ICSE 2012)*. (Zurich, Switzerland, June 2–9, 2012). Ed. by M. Glinz, G. C. Murphy, and M. Pezzè. IEEE, 2012, pp. 419–429. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227173.

[BHB99] I. T. Bowman, R. C. Holt, and N. V. Brewster. "Linux as a case study: its extracted software architecture". In: *1999 International Conference on Software Engineering (ICSE 1999)*. (Los Angeles, USA, May 16–22, 1999). Ed. by B. W. Boehm, D. Garlan, and J. Kramer. ACM, 1999, pp. 555–563. DOI: 10.1145/302405.302691.

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*. Vol. 4334. LNAI. Springer-Verlag, 2007. ISBN: 978-3-540-68977-5. DOI: 10.1007/978-3-540-69061-0.

[BHW06] G. Beuster, N. Henrich, and M. Wagner. "Real world verification – Experiences from the Verisoft email client". In: *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006)*. (Seattle, USA, Aug. 21, 2006). Ed. by G. Sutcliffe, R. Schmidt, and S. Schulz. Vol. 192. CEUR Workshop Proceedings. CEUR-WS.org, Aug. 2006, pp. 112–125.

[BK06] B. Beckert and V. Klebanov. "Must program verification systems and calculi be verified?" In: *3rd International Verification Workshop (VERIFY 2006), Workshop at Federated Logic Conferences (FLoC)*. (Seattle, Washington, Aug. 15–16, 2006). 2006, pp. 34–41.

[BLS05] M. Barnett, K. R. M. Leino, and W. Schulte. "The Spec# programming system: An overview". In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Ed. by G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean. Vol. 3362. LNCS. Springer, 2005, pp. 49–69. ISBN: 978-3-540-24287-1. DOI: 10.1007/978-3-540-30569-9_3.

[BLW08] S. Böhme, K. R. M. Leino, and B. Wolff. "HOL-Boogie—An interactive prover for the Boogie program-verifier". In: *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*. (Montreal, Canada, Aug. 18–21, 2008). Ed. by O. A. Mohamed, C. A. Muñoz, and S. Tahar. Vol. 5170. LNCS. Springer, 2008, pp. 150–166. DOI: 10.1007/978-3-540-71067-7_15.

[BM11] S. Böhme and M. Moskal. *Fat pointers, skinny annotations: A heap model for modular C verification*. 2011.

[Bog08] S. Bogan. "Formal Specification of a Simple Operating System". PhD thesis. Saarland University, Computer Science Department, Aug. 2008. URL: http://www-wjp.cs.uni-sb.de/publikationen/Bog08.pdf.

[Böh09] S. Böhme. "Proof reconstruction for Z3 in Isabelle/HOL". In: *7th International Workshop on Satisfiability Modulo Theories (SMT 2009)*. (Montréal, Canada, Aug. 2–3, 2009). 2009.

[Böh12] S. Böhme. "Proving Theorems of Higher-Order Logic with SMT Solvers". PhD thesis. Technische Universität München, 2012.

[BST99] M. Bidoit, D. Sannella, and A. Tarlecki. "Architectural specifications in CASL". In: *7th International Conference on Algebraic Methodology and Software Technology (AMAST 1999)*. Ed. by A. M. Haeberer. Vol. 1548. LNCS. Springer, 1999, pp. 341–357. ISBN: 978-3-540-65462-9. DOI: 10.1007/3-540-49253-4_25.

[BW10] T. Bormer and M. Wagner. "Towards testing a verifying compiler". In: *Formal Verification of Object-Oriented Software, Papers presented at the International Conference*. (Paris, France, June 28–30, 2010). Ed. by B. Beckert and C. Marché. Vol. 2010-13. Karlsruhe Reports in Informatics. Karlsruhe Institute of Technology, Technical Report, 2010.

[CASL] *Common Algebraic Specification Language (CASL)*. 2008. URL: http://www.informatik.uni-bremen.de/cofi/wiki/index.php/CASL (visited on 07/17/2014).

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda. "A tool for checking ANSI-C programs". In: *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. (Barcelona, Spain, Mar. 29–Apr. 2, 2004). Ed. by K. Jensen and A. Podelski. Vol. 2988. LNCS. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[Cla+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided abstraction refinement". In: *12th International Conference on Computer Aided Verification (CAV 2000)*. (Chicago, USA, July 15–19, 2000). Ed. by E. A. Emerson and A. P. Sistla. Vol. 1855. LNCS. Springer, 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[CLANG] *clang: A C language family frontend for LLVM*. URL: http://clang.llvm.org/ (visited on 06/30/2014).

[Cod14] Codenomicon. *The Heartbleed Bug*. 2014. URL: http://www.heartbleed.com (visited on 07/09/2014).

[Coh+09a] E. Cohen, E. Alkassar, V. Boyarinov, M. Dahlweid, U. Degenbaev, M. A. Hillebrand, B. Langenstein, D. Leinenbach, M. Moskal, S. Obua, W. J. Paul, H. Pentchev, E. Petrova, T. Santen, N. Schirmer, S. Schmaltz, W. Schulte, A. Shadrin, S. Tobies, A. Tsyban, and S. Tverdyshev. "Invariants, modularity, and rights". In: *7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI 2009)*. (Novosibirsk, Russia, June 15–19, 2009). Ed. by A. Pnueli, I. Virbitskaite, and A. Voronkov. Vol. 5947. LNCS. Springer, 2009, pp. 43–55. DOI: 10.1007/978-3-642-11486-1_4.

[Coh+09b] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. "VCC: A practical system for verifying concurrent C". In: *Theorem Proving in Higher Order Logics (TPHOLs 2009)*. (Munich, Germany, Aug. 17–20, 2009). Vol. 5674. LNCS. Springer, 2009, pp. 23–42. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_2.

[Coh+09c] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. *A practical verification methodology for concurrent programs*. MSR-TR-2009-15. 2009. URL: http://research.microsoft.com/en-us/um/people/moskal/pdf/concurrency3.pdf (visited on 02/04/2014).

[Coh+12] E. Cohen, M. A. Hillebrand, M. Moskal, S. Tobies, and W. Schulte. *Verifying C Programs: A VCC Tutorial*. Version 0.2. Aug. 25, 2012. URL: http://www.codeplex.com/Download?ProjectName=VCC&DownloadId=476508 (visited on 02/02/2014).

[Coo78] S. A. Cook. "Soundness and completeness of an axiom system for program verification". In: *SIAM Journal of Computing* 7.1 (1978), pp. 70–90. DOI: 10.1137/0207005.

[CPS13] E. Cohen, W. Paul, and S. Schmaltz. "Theory of multi core hypervisor verification". In: *39th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2013)*. (Špindlerův Mlýn, Czech Republic, Jan. 26–31, 2013). Ed. by P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. Nawrocki, and H. Sack. Vol. 7741. LNCS. Springer, 2013, pp. 1–27. DOI: 10.1007/978-3-642-35843-2_1.

[CS10] E. Cohen and B. Schirmer. "From total store order to sequential consistency: A practical reduction theorem". In: *First International Conference on Interactive Theorem Proving (ITP 2010)*. (Edinburgh, UK, June 11–14, 2010). Ed. by M. Kaufmann and L. C. Paulson. Vol. 6172. LNCS. Springer, 2010, pp. 403–418. DOI: 10.1007/978-3-642-14052-5_28.

[Cuo+12a] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A software analysis perspective". In: *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*. (Thessaloniki, Greece, Oct. 1–5, 2012). Vol. 7504. LNCS. Springer, 2012, pp. 233–247. ISBN: 978-3-642-33825-0. DOI: 10.1007/978-3-642-33826-7_16.

[Cuo+12b] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. "Testing static analyzers with randomly generated programs". In: *NASA Formal Methods: 4th International Symposium (NFM 2012)*. (Norfolk, USA, Apr. 3–5, 2012). Ed. by A. Goodloe and S. Person. Vol. 7226. LNCS. Springer, 2012, pp. 120–125. ISBN: 978-3-642-28890-6. DOI: 10.1007/978-3-642-28891-3_12.

[Dah+09] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. "VCC: Contract-based modular verification of concurrent C". In: *31st International Conference on Software Engineering (ICSE 2009)*. (Vancouver, Canada, May 16–24, 2009). IEEE, May 2009, pp. 429–430. DOI: 10.1109/ICSE-COMPANION.2009.5071046.

[DL05] R. DeLine and K. R. M. Leino. *BoogiePL: A typed procedural language for checking object-oriented programs*. Tech. rep. MSR-TR-2005-70. Microsoft Research, 2005. URL: `ftp://ftp.research.microsoft.com/pub/tr/TR-2005-70.pdf`.

[Doe11] T. W. Doeppner. *Operating Systems In Depth: Design and Programming*. Wiley, 2011. ISBN: 978-0-471-68723-8.

[Dör10] J. Dörrenbächer. “Formal Specification and Verification of a Microkernel”. PhD thesis. Saarland University, Saarbrücken, 2010. URL: `http://www-wjp.cs.uni-saarland.de/publikationen/JD10.pdf`.

[Dov+10] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. “Lazy behavioral subtyping”. In: *Journal of Logic and Algebraic Programming* 79.7 (2010), pp. 578–607. DOI: `10.1016/j.jlap.2010.07.008`.

[DSS09] M. Daum, N. Schirmer, and M. Schmidt. “Implementation correctness of a real-time operating system”. In: *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009)*. (Hanoi, Vietnam, Nov. 23–27, 2009). Ed. by D. V. Hung and P. Krishnan. IEEE, 2009, pp. 23–32. DOI: `10.1109/SEFM.2009.14`.

[Dup13] F. Dupressoir. “Proving Cryptographic C Programs Secure with General-Purpose Verification Tools”. PhD thesis. The Open University, 2013.

[FBL10] M. Fähndrich, M. Barnett, and F. Logozzo. “Embedded contract languages”. In: *Proceedings of the ACM Symposium on Applied Computing (SAC 2010)*. (Sierre, Switzerland, Mar. 22–26, 2010). Ed. by S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C. Hung. ACM, 2010, pp. 2103–2110. DOI: `10.1145/1774088.1774531`.

[Fin13] J. Finkle. “Adobe data breach more extensive than previously disclosed”. In: *Reuters. (Online Article)* (Oct. 2013). URL: `http://www.reuters.com/article/2013/10/29/us-adobe-cyberattack-idUSBRE99S1DJ20131029`.

[Flo67] R. Floyd. “Assigning meaning to programs”. In: *Mathematical Aspects of Computer Science*. Ed. by J. T. Schwartz. Proceedings of Symposia in Applied Mathematics 19. American Mathematical Society, 1967, pp. 19–32.

[FM07] J.-C. Filliâtre and C. Marché. “The Why/Krakatoa/Caduceus platform for deductive program verification”. In: *19th International Conference on Computer Aided Verification (CAV 2007)*. Ed. by W. Damm and H. Hermanns. Vol. 4590. LNCS. Springer, 2007, pp. 173–177. ISBN: 978-3-540-73367-6. DOI: `10.1007/978-3-540-73368-3_21`.

[FN79] R. J. Feiertag and P. G. Neumann. “The foundations of a provably secure operating system (PSOS)”. In: *Proceedings of the National Computer Conference*. New York, USA: AFIPS Press, 1979, pp. 329–334.

[Fre03] Freescale Semiconductor. *G2 PowerPC™ Core Reference Manual*. 1st. Freescale Semiconductor, June 2003. URL: `www.freescale.com/files/32bit/doc/ref_manual/G2CORERM.pdf`.

[Fre05] Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture*. 2005. URL: www.freescale.com/files/product/doc/MPCFPE32B.pdf.

[Gar+05] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. "On the correctness of operating system kernels". In: *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*. Ed. by J. Hurd and T. Melham. Vol. 3603. LNCS. Springer, 2005, pp. 1–16. ISBN: 978-3-540-28372-0. DOI: 10.1007/11541868_1.

[GD07] V. Ganesh and D. L. Dill. "A decision procedure for bit-vectors and arrays". In: *19th International Conference on Computer Aided Verification, (CAV 2007)*. (Berlin, Germany, July 3–7, 2007). Ed. by W. Damm and H. Hermanns. Vol. 4590. LNCS. Springer, 2007, pp. 519–531. DOI: 10.1007/978-3-540-73368-3_52.

[GKL04] A. Groce, D. Kroening, and F. Lerda. "Understanding counterexamples with explain". In: *16th International Conference on Computer Aided Verification (CAV 2004)*. (Boston, USA, July 13–17, 2004). Ed. by R. Alur and D. Peled. Vol. 3114. LNCS. Springer, 2004, pp. 453–456. DOI: 10.1007/978-3-540-27813-9_35.

[Gla01] R. van Glabbeek. "The linear time - branching time spectrum I. The semantics of concrete, sequential processes". In: *Handbook of Process Algebra*. Ed. by J. Bergstra, A. Ponse, and S. Smolka. Elsevier, 2001, pp. 3–99. ISBN: 978-0-444-82830-9. DOI: 10.1016/B978-044482830-9/50019-9.

[GLM11] C. L. Goues, K. R. M. Leino, and M. Moskal. "The Boogie verification debugger (tool paper)". In: *9th International Conference on Software Engineering and Formal Methods (SEFM 2011)*. (Montevideo, Uruguay, Nov. 14–18, 2011). Ed. by G. Barthe, A. Pardo, and G. Schneider. Vol. 7041. LNCS. Springer, 2011, pp. 407–414. DOI: 10.1007/978-3-642-24690-6_28.

[Gre12] S. Grebing. "Evaluating and Improving the Usability of Interactive Verification Systems". Diplomarbeit. Universität Koblenz-Landau, Aug. 2012.

[Har79] D. Harel. *First-Order Dynamic Logic*. Vol. 68. LNCS. Springer, 1979. ISBN: 3-540-09237-4. DOI: 10.1007/3-540-09237-4.

[HL09] M. A. Hillebrand and D. C. Leinenbach. "Formal verification of a reader-writer lock implementation in C". In: *Proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009)*. Vol. 254. Electronic Notes in Theoretical Computer Science. Elsevier, Oct. 2009, pp. 123–141. DOI: 10.1016/j.entcs.2009.09.063.

[Hoa03] T. Hoare. "The verifying compiler: A grand challenge for computing research". In: *Modular Programming Languages*. Ed. by L. Böszörményi and P. Schojer. Vol. 2789. LNCS. Springer, 2003, pp. 25–35. ISBN: 978-3-540-40796-6. DOI: 10.1007/978-3-540-45213-3_4.

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[IP08] T. In der Rieden and W. J. Paul. "Beweisen als Ingenieurwissenschaft: Verbundprojekt Verisoft (2003–2007)". In: *Informatikforschung in Deutschland*. Ed. by B. Reuse and R. Vollmar. Springer, 2008, pp. 321–326. ISBN: 978-3-540-76549-3. DOI: 10.1007/978-3-540-76550-9.

[ISO+11] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 9899:2011 Information technology – Programming languages – C*. Geneva, Switzerland, Dec. 8, 2011.

[Jac06] D. Jackson. *Software Abstractions – Logic, Language, and Analysis*. MIT Press, 2006, pp. I–XVI, 1–350. ISBN: 978-0-262-10114-1.

[Jon83] C. B. Jones. "Specification and design of (parallel) programs". In: *Proceedings of the IFIP Congress* (1983), pp. 321–332.

[Jon91] B. Jonsson. "Simulations between specifications of distributed systems". In: *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR 1991)*. (Amsterdam, The Netherlands, Aug. 26–29, 1991). Ed. by J. C. M. Baeten and J. F. Groote. Vol. 527. LNCS. Springer, 1991, pp. 346–360. ISBN: 3-540-54430-5. DOI: 10.1007/3-540-54430-5_99.

[Jon97] D. Jones. *Who Guards the Guardians?* Feb. 1997. URL: http://www.knosof.co.uk/whoguard.html.

[JP04] B. Jacobs and E. Poll. "Java program verification at Nijmegen: Developments and perspective". In: *Software Security – Theories and Systems, Second Mext-NSF-JSPS International Symposium (ISSS 2003)*. (Tokyo, Japan, Nov. 4–6, 2003). Ed. by K. Futatsugi, F. Mizoguchi, and N. Yonezaki. Vol. 3233. LNCS. Springer, 2004, pp. 134–153. ISBN: 978-3-540-23635-1. DOI: 10.1007/978-3-540-37621-7_7.

[JP08] B. Jacobs and F. Piessens. *The VeriFast program verifier*. Tech. rep. CW-520. Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008. URL: http://people.cs.kuleuven.be/~bart.jacobs/verifast/verifast.pdf.

[Kle+10] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. "seL4: Formal verification of an operating system kernel". In: *Communications of the ACM* 53.6 (June 2010), pp. 107–115. DOI: 10.1145/1743546.1743574.

[Kle+11] V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. "The 1st Verified Software Competition: Experience report". In: *17th International Symposium on Formal*

Methods (FM 2010). (Edinburgh, UK, Aug. 18, 2010). Ed. by M. Butler and W. Schulte. Vol. 6664. LNCS. Materials available at `www.vscomp.org`. Springer, 2011. DOI: `10.1007/978-3-642-21437-0_14`.

[Kle+14] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. "Comprehensive formal verification of an OS microkernel". In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), 2:1–2:70. DOI: `10.1145/2560537`.

[Kle09] G. Klein. "Operating system verification – An overview". In: *Sādhanā* 34.1 (2009), pp. 27–69. ISSN: 0256-2499. DOI: `10.1007/s12046-009-0002-4`.

[Kle14] G. Klein. "Proof engineering considered essential". In: *19th International Symposium on Formal Methods (FM 2014)*. (Singapore, May 12–16, 2014). Ed. by C. B. Jones, P. Pihlajasaari, and J. Sun. Vol. 8442. LNCS. Springer, 2014, pp. 16–21. DOI: `10.1007/978-3-319-06410-9_2`.

[Kup06] O. Kupferman. "Sanity checks in formal verification". In: *17th International Conference on Concurrency Theory (CONCUR 2006)*. (Bonn, Germany, Aug. 27–30, 2006). Ed. by C. Baier and H. Hermanns. Vol. 4137. LNCS. Springer, 2006, pp. 37–51. DOI: `10.1007/11817949_3`.

[KW07] R. Kaiser and S. Wagner. "Evolution of the PikeOS microkernel". In: *1st International Workshop on Microkernels for Embedded Systems (MIKES 2007)*. Ed. by I. Kuz and S. M. Petters. National ICT Australia, 2007.

[KWG04] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. "Software fault interactions and implications for software testing". In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 418–421. ISSN: 0098-5589. DOI: `10.1109/TSE.2004.24`.

[LA04] C. Lattner and V. S. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004)*. (San Jose, USA, Mar. 20–24, 2004). IEEE Computer Society, 2004, pp. 75–88. DOI: `10.1109/CGO.2004.1281665`.

[Lei08] D. C. Leinenbach. "Compiler Verification in the Context of Pervasive System Verification". PhD thesis. Saarland University, Saarbrücken, July 2008. URL: `http://www-wjp.cs.uni-saarland.de/publikationen/Lei08.pdf`.

[Lei12] K. R. M. Leino. "Automating induction with an SMT solver". In: *13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*. Ed. by V. Kuncak and A. Rybalchenko. Vol. 7148. LNCS. Springer, 2012, pp. 315–331. ISBN: 978-3-642-27939-3. DOI: `10.1007/978-3-642-27940-9_21`.

[Lei13] K. R. M. Leino. “Automating theorem proving with SMT”. In: *4th International Conference on Interactive Theorem Proving (ITP 2013)*. (Rennes, France, July 22–26, 2013). Ed. by S. Blazy, C. Paulin-Mohring, and D. Pichardie. Vol. 7998. LNCS. Springer, 2013, pp. 2–16. DOI: 10.1007/978-3-642-39634-2_2.

[Lio+96] J.-L. Lions et al. *Ariane 5 flight 501 failure*. Report by the enquiry board. July 1996. URL: http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[LM10] K. R. M. Leino and M. Moskal. “Usable auto-active verification”. In: *Usable Verification workshop*. (Redmond, USA, Nov. 15–16, 2010). 2010. URL: http://fm.csl.sri.com/UV10.

[LNT12] T. Liu, M. Nagel, and M. Taghdiri. “Bounded program verification using an SMT solver: A case study”. In: *5th IEEE International Conference on Software Testing, Verification and Validation (ICST 2012)*. (Montreal, Canada, Apr. 14–21, 2012). Ed. by G. Antoniol, A. Bertolino, and Y. Labiche. IEEE Computer Society, 2012, pp. 101–110. DOI: 10.1109/ICST.2012.90.

[Loc12] A. Lochbihler. “A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler”. PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012. DOI: 10.5445/KSP/1000028867.

[Log13] F. Logozzo. “Practical specification and verification with code contracts”. In: *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology (HILT 2013)*. (Pittsburgh, USA, Nov. 10–14, 2013). Ed. by J. Boleng and S. T. Taft. ACM, 2013, pp. 7–8. ISBN: 978-1-4503-2467-0. DOI: 10.1145/2527269.2534188.

[LQL12] A. Lal, S. Qadeer, and S. K. Lahiri. “A solver for reachability modulo theories”. In: *24th International Conference on Computer Aided Verification (CAV 2012)*. (Berkeley, USA, July 7–13, 2012). Ed. by P. Madhusudan and S. A. Seshia. Vol. 7358. LNCS. Springer, 2012, pp. 427–443. DOI: 10.1007/978-3-642-31424-7_32.

[LS09] D. Leinenbach and T. Santen. “Verifying the Microsoft Hyper-V hypervisor with VCC”. In: *Second World Congress on Formal Methods (FM 2009)*. (Eindhoven, The Netherlands, Nov. 2–6, 2009). Ed. by A. Cavalcanti and D. Dams. Vol. 5850. LNCS. Springer, 2009, pp. 806–809. DOI: 10.1007/978-3-642-05089-3_51.

[LT93] N. Leveson and C. Turner. “An investigation of the Therac-25 accidents”. In: *Computer* 26.7 (July 1993), pp. 18–41. ISSN: 0018-9162. DOI: 10.1109/MC.1993.274940.

[LV95] N. A. Lynch and F. W. Vaandrager. “Forward and backward simulations: I. Untimed systems”. In: *Information and Computation* 121.2 (1995), pp. 214–233. ISSN: 0890-5401. DOI: 10.1006/inco.1995.1134.

[Mac03] P. Mackerras. "Low-level optimizations in the PowerPC/Linux kernels". In: *Proceedings of the Linux Symposium*. (Ottawa, Canada, July 23–26, 2003). 2003, pp. 304–314.

[MB08] L. de Moura and N. Bjørner. "Z3: An efficient SMT solver". In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. (Budapest, Hungary, Mar. 29–Apr. 6, 2008). Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. LNCS. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MFS12] F. Merz, S. Falke, and C. Sinz. "LLBMC: Bounded model checking of C and C++ programs using a compiler IR". In: *4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012)*. (Philadelphia, USA, Jan. 28–29, 2012). Ed. by R. Joshi, P. Müller, and A. Podelski. Vol. 7152. LNCS. Springer, 2012, pp. 146–161. DOI: 10.1007/978-3-642-27705-4_12.

[Mil71] R. Milner. "An algebraic definition of simulation between programs". In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI 1971)*. (London, England). Ed. by D. C. Cooper. Morgan Kaufmann Publishers Inc., 1971, pp. 481–489.

[Mil83] R. Milner. "Calculi for synchrony and asynchrony". In: *Theoretical Computer Science* 25.3 (1983), pp. 267–310. ISSN: 0304-3975. DOI: 10.1016/0304-3975(83)90114-7.

[Mit+13] A. Mittal, D. Bansal, S. Bansal, and V. Sethi. "Efficient virtualization on embedded Power Architecture® platforms". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*. (Houston, USA, Mar. 16–20, 2013). Ed. by V. Sarkar and R. Bodík. ACM, 2013, pp. 445–458. DOI: 10.1145/2451116.2451163.

[MITR14] MITRE. *Common Vulnerabilities and Exposures entry for TLS heartbeat read overrun (CVE-2014-0160)*. 2014. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[MM13] C. Marché and Y. Moy. *The Jessie plugin for deductive verification in Frama-C—Tutorial and reference manual*. At http://krakatoa.lri.fr/jessie.pdf. 2013.

[Mos04] P. D. Mosses. *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*. Vol. 2960. LNCS. Springer, 2004. ISBN: 978-3-540-21301-7. DOI: 10.1007/b96103.

[Mos09] M. Moskal. "Programming with triggers". In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT 2009)*. (Montreal, Canada). ACM, 2009, pp. 20–29. ISBN: 978-1-60558-484-3. DOI: 10.1145/1670412.1670416.

[MR11] P. Müller and J. N. Ruskiewicz. "Using debuggers to understand failed verification attempts". In: *17th International Symposium on Formal Methods (FM 2011)*. Ed. by M. Butler and W. Schulte. Vol. 6664. LNCS. Springer, 2011, pp. 73–87. ISBN: 978-3-642-21436-3. DOI: 10.1007/978-3-642-21437-0_8.

[Mur+12] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. "Noninterference for operating system kernels". In: *Second International Conference on Certified Programs and Proofs (CPP 2012)*. (Kyoto, Japan, Dec. 13–15, 2012). Ed. by C. Hawblitzel and D. Miller. Vol. 7679. LNCS. Springer, 2012, pp. 126–142. DOI: 10.1007/978-3-642-35308-6_12.

[NG14] NICTA and General Dynamics C4 Systems. *seL4 Microkernel*. 2014. URL: http://sel4.systems/ (visited on 08/29/2014).

[NIC11a] NICTA. *L4.verified Proof Statistics*. 2011. URL: http://ertos.nicta.com.au/research/l4.verified/numbers.pml (visited on 01/22/2014).

[NIC11b] NICTA. *The L4.verified project*. 2011. URL: http://ertos.nicta.com.au/research/l4.verified/ (visited on 08/29/2014).

[NWP02] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002. ISBN: 978-3-540-45949-1. DOI: 10.1007/3-540-45949-9.

[Ohe01] D. von Oheimb. "Hoare logic for Java in Isabelle/HOL". In: *Concurrency and Computation: Practice and Experience* 13.13 (2001), pp. 1173–1214. DOI: 10.1002/cpe.598.

[OSSL14] The OpenSSL Software Foundation. *OpenSSL Project*. URL: http://www.openssl.org/ (visited on 07/07/2014).

[Par81] D. M. R. Park. "Concurrency and automata on infinite sequences". In: *5th GI-Conference on Theoretical Computer Science*. (Karlsruhe, Germany, Mar. 23–25, 1981). Ed. by P. Deussen. Vol. 104. LNCS. Springer, 1981, pp. 167–183. DOI: 10.1007/BFb0017309.

[PFW13] N. Polikarpova, C. A. Furia, and S. West. "To run what no one has run before: Executing an intermediate verification language". In: *4th International Conference on Runtime Verification (RV 2013)*. (Rennes, France, Sept. 24–27, 2013). Ed. by A. Legay and S. Bensalem. Vol. 8174. LNCS. Springer, 2013, pp. 251–268. DOI: 10.1007/978-3-642-40787-1_15.

[PG74] G. J. Popek and R. P. Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421. DOI: 10.1145/361011.361073.

[PSS12] W. J. Paul, S. Schmaltz, and A. Shadrin. "Completing the automated verification of a small hypervisor – Assembler code verification". In: *10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*. (Thessaloniki, Greece, Oct. 1–5, 2012). Ed. by G. Eleftherakis, M. Hinchey, and M. Holcombe. Vol. 7504. LNCS. Springer, 2012, pp. 188–202. DOI: 10.1007/978-3-642-33826-7_13.

[RCT12] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2012. ISBN: 3-900051-07-0. URL: http://www.R-project.org.

[RE12] RTCA SC-205 and EUROCAE WG-12. *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. Washington: Radio Technical Commission for Aeronautics (RTCA), Inc., Jan. 2012.

[RE92] RTCA SC-167 and EUROCAE WG-12. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Washington: Radio Technical Commission for Aeronautics (RTCA), Inc., Dec. 1992.

[SFM10] C. Sinz, S. Falke, and F. Merz. "A precise memory model for low-level bounded model checking". In: *5th International Workshop on Systems Software Verification (SSV 2010)*. (Vancouver, Canada, Oct. 6–7, 2010). Ed. by R. Huuck, G. Klein, and B. Schlich. USENIX Association, 2010. URL: https://www.usenix.org/conference/ssv10/precise-memory-model-low-level-bounded-model-checking.

[SGG08] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. 8th. Wiley, 2008. ISBN: 978-0-470-23399-3.

[She11] I. Sherr. "Hackers breach second Sony service". In: *Wall Street Journal Online* (May 2011).

[SMK13] T. A. L. Sewell, M. O. Myreen, and G. Klein. "Translation validation for a verified OS kernel". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. (Seattle, USA, June 16–19, 2013). Ed. by H. Boehm and C. Flanagan. ACM, 2013, pp. 471–482. DOI: 10.1145/2462156.2462183.

[SN05] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1-55860-910-5.

[SYSa] SYSGO. *PikeOS embedded virtualization*. URL: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/embedded-virtualization/ (visited on 10/01/2013).

[SYSb] SYSGO. *PikeOS Hypervisor – Hardware support*. URL: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/hardware-support/ (visited on 09/04/2014).

[SYSc] SYSGO. *PikeOS Hypervisor – Safety architecture*. URL: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/safety-architecture/ (visited on 09/03/2014).

[SYSd] SYSGO. *Safety & security certification for single- and multi-core CPUs – RTCA DO-178B*. URL: http://www.sysgo.com/services-solutions/safety-security-certification/rtca-do-178b/ (visited on 09/03/2014).

[SYS08] SYSGO. *AIRBUS selects SYSGO's PikeOS as DO-178B reference platform for the A350 XWB*. Nov. 2008. URL: http://www.sysgo.com/news-events/press/press/details/article/airbus-selects-sysgos-pikeos-as-do-178b-reference-platform-for-the-a350-xwb/.

[SYS14] SYSGO AG. *PikeOS Hypervisor*. 2014. URL: http://www.sysgo.com.

[TH08] N. Tillmann and J. de Halleux. "Pex – White box test generation for .NET". In: *2nd International Conference on Tests and Proofs (TAP 2008)*. (Prato, Italy, Apr. 9–11, 2008). Ed. by B. Beckert and R. Hähnle. Vol. 4966. LNCS. Springer, 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9_10.

[Tsc+11] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. "Verifying Eiffel programs with Boogie". In: *CoRR* abs/1106.4700 (2011).

[Tsc+13] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. "Program checking with less hassle". In: *5th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2013)*. (Menlo Park, USA, May 17–19, 2013). Ed. by E. Cohen and A. Rybalchenko. Vol. 8164. LNCS. Springer, 2013, pp. 149–169. DOI: 10.1007/978-3-642-54108-7_8.

[Tsy09] A. Tsyban. "Formal Verification of a Framework for Microkernel Programmers". PhD thesis. Saarland University, Saarbrücken, 2009. URL: http://www-wjp.cs.uni-saarland.de/publikationen/Tsy09.pdf.

[Uri00] T. E. Uribe. "Combinations of model checking and theorem proving". In: *Frontiers of Combining Systems, Third International Workshop (FroCoS 2000)*. (Nancy, France, Mar. 22–24, 2000). Ed. by H. Kirchner and C. Ringeissen. Vol. 1794. LNCS. Springer, 2000, pp. 151–170. DOI: 10.1007/10720084_11.

[Van+08] D. Vanoverberghe, N. Bjørner, J. de Halleux, W. Schulte, and N. Tillmann. "Using dynamic symbolic execution to improve deductive verification". In: *15th International SPIN Workshop on Model Checking Software*. (Los Angeles, USA, Aug. 10–12, 2008). Ed. by K. Havelund, R. Majumdar, and J. Palsberg. Vol. 5156. LNCS. Springer, 2008, pp. 9–25. DOI: 10.1007/978-3-540-85114-1_4.

[VCC12] *The VCC Manual*. Version 0.2. Working draft. Aug. 2012. URL: https://www.codeplex.com/Download?ProjectName=VCC&DownloadId=476509 (visited on 07/20/2014).

[Ver07] Verisoft Consortium. *Verisoft XT Projektzusammenfassung*. 2007. URL: http://www.verisoft.de/.rsrc/StartSeite/zusammenfassung_verisoftxt.pdf.

[Ver10] Verisoft Consortium. *The Verisoft XT Project*. 2007–2010. URL: http://www.verisoftxt.de.

[Wag09] M. Wagner. "Testing a Verification Environment". Diplomarbeit. Universität Koblenz-Landau, Nov. 2009.

[WKP80] B. J. Walker, R. A. Kemmerer, and G. J. Popek. “Specification and verification of the UCLA Unix security kernel”. In: *Communications of the ACM* 23.2 (Feb. 1980), pp. 118–131. ISSN: 0001-0782. DOI: 10.1145/358818.358825.

[Wri94] J. von Wright. “The formal verification of a proof checker”. SRI internal report. 1994.

[Yan+11] X. Yang, Y. Chen, E. Eide, and J. Regehr. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*. (San Jose, USA, June 4–8, 2011). Ed. by M. W. Hall and D. A. Padua. ACM, 2011, pp. 283–294. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532.

[Zee+07] K. Zee, V. Kuncak, M. Taylor, and M. C. Rinard. “Runtime checking for program verification”. In: *7th International Workshop on Runtime Verification (RV 2007)*. (Vancouver, Canada, Mar. 13, 2007). Ed. by O. Sokolsky and S. Tasiran. Vol. 4839. LNCS. Springer, 2007, pp. 202–213. DOI: 10.1007/978-3-540-77395-5_17.

[ZKR08] K. Zee, V. Kuncak, and M. Rinard. “Full functional verification of linked data structures”. In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. (Tucson, AZ, USA, June 7–13, 2008). ACM, 2008, pp. 349–361. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375624.