OS Verification Extended

On the Formal Verification of Device Drivers and the Correctness of Client/Server Software

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Eyad Alkassar

eyad@wjpserver.cs.uni-sb.de

Saarbrücken, Juli 2009

Tag des Kolloqiums:	17.07.2009
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Kurt Mehlhorn
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Gert Smolka
3. Berichterstatter:	Prof. Dr. Andreas Podelski
akademischer Mitarbeiter:	Dr. Mark A. Hillebrand

Mama wa Baba — Schukran.

Danksagung

An dieser Stelle möchte ich all jenen danken, die zum Gelingen der vorliegenden Arbeit beigetragen haben.

Zunächst gilt mein Dank meiner Familie; meiner Mutter und meinem Vater, denen ich fast alles zu Verdanken habe, und meinen Geschwistern Ammar, Muhannad und Manar, auf die immer Verlass ist.

Meinem Mentor Herrn Prof. Paul danke ich ganz besonders für die Möglichkeit, meine Promotion im Rahmen eines so spannenden Projektes wie Verisoft durchführen zu können und für die wissenschaftliche Betreuung der Arbeit.

Mein Dank gilt auch meinen (ehemaligen und derzeitigen) Arbeitskollegen am Lehrstuhl von Herrn Paul. Mark (der lokales-lernen erbende-wen-wurst), Dirk (der allzeit Hilfsbereite), Norbert (der ehrenwerte Denker) und Tom (der trickreiche Menschenkenner) waren eine unschätzbare Bereicherung. Meinem Freund Steffen möchte ich ganz besonders danken für die gemeinsamen Jahre im Raum 318. Ausdrücklich danken für die gemeinsame Zeit möchte ich auch Hristo, Sebastian, Sergey, Andrey und Peter.

Schließlich bleibt mir meinem Freund Aref zu danken, dessen weiser Rat in manch schwerer Stunde unendlich wertvoll war.

This thesis work was funded by DFG Graduiertenkolleg 'Leistungsgarantien für Rechnersysteme' and by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Abstract

This thesis tackles two important challenges in OS verification: The formal verification of device drivers and the correctness of client/server software.

Device drivers are an integral part of system software. Not only high-level functionality such as file I/O depends on devices. Even basic OS features, such as demand paging, need correctly implemented drivers. In this thesis, we show how to pervasively integrate devices and their drivers into a language stack reaching from the level of assembly up to high-level languages. This stack is leveraged for the formal verification of a simple hard disk driver, which is subsequently embedded into Verisoft's micro kernel. To the best of our knowledge, this marks the first formal functional verification of a device driver against a realistic device and system model.

Remote procedure calls (RPCs) lie at the heart of any client/server software. In the second part of this thesis, we present a specification of an RPC mechanism and we outline how to verify an implementation of this mechanism at the code level. The formalization is based on a model of user processes running concurrently under a simple OS, which provides inter-process communication and portmapper system calls. A simple theory of non interference permits us to use conventional sequential program analysis between system calls. To the best of our knowledge this is the first treatment of the correctness of an entire RPC mechanism at the code level.

Zusammenfassung

Diese Arbeit behandelt zwei wichtige Probleme in der Verifikation von Betriebssystemen (BS): Die formale Verifikation von Gerätetreibern und die Korrektheit von Client/Server Software.

Grundlegende Funktionen eines BS, wie z.B. Demand Paging, setzen korrekt implementierte Treiber voraus. In dieser Arbeit zeigen wir auf, wie Geräte nahtlos in allen Semantikschichten integriert werden können — von Assembler bis hin zu einer C ähnlichen Hochsprache. Diese durchgängige Theorie wird anschließend verwendet, um einen einfachen Festplattentreiber (Teil des Verisoft Mikrokerns) formal zu verifizieren. So weit uns bekannt, stellt dies die erste formale Verifikation eines Treibers im Kontext eines realistischen Geräte- und Systemmodells dar.

Implementierungen von Client/Server Software basieren oftmals auf Remote Procedure Calls (RPCs). Im zweiten Teil dieser Arbeit, spezifizieren wir einen solchen RPC Mechanismus und skizzieren dessen Verifikation auf Codeebene. Die Formalisierung basiert auf einem Modell von Benutzerprozessen die nebenläufig in einem einfachen BS ausgeführt werden. Dieses BS stellt Interprozess-Kommunikation und Portmapper Funktionalität über spezielle Systemaufrufe zur Verfügung. Um sequentiell über einzelne Prozesse argumentieren zu können, führen wir eine kleine Theorie zur Bestimmung der Abhängigkeit von Systemaufrufen ein. So weit uns bekannt, behandelt diese Arbeit erstmals die Korrektheit eines vollständigen RPC Mechanismus auf Codeebene.

Contents

Chapter 1

Introduction

The heart beat of our society If technology is labeled the new religion of our modern society, software developers should be called its high priests. In almost all niches of our daily life we are at their mercy, believing in the infallibility of their products: When we drive to work with the car, dozens of programs — from the entertainment system to the airbag control — assist us and ensure that we reach our destination safely. When we travel around the world with the airplane, most of the time a program is flying. Future presidents and rulers are elected at black boxes, in which a program counts the vote. And finally, when we get old, even the beat of our heart is controlled by a piece of software running in the pacemaker.

Even though a single error in a single code line may destroy hundreds of lives, society responds surprisingly calm. From where comes this unbounded *trust*? There are four ways to deal with the threat of software errors.

Followers of the first way — lets call them traditionalists — claim that it worked out in the past, so why shall we bother about the future? This argument has three major flaws. Firstly, it is impudent to plead on the record of a technology which did not exist two generations ago. Since then, the tremendous increase of the application domain of computer systems has been probably singular in the history of technology. Secondly, it did *not* work out so far; there is already an alarming record of catastrophes caused by bugs in programs. Here, only one example for illustration: The death of at least five people could be directly ascribed to a bug in the code controlling the Therac-25 radiation therapy machine when it administered excessive quantities of X-rays [Lev93]. In 2002 a study of the US National Institute of Standards and Technology came to the result that '*software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product*'. And, thirdly, even if we ignore these facts, a single worst-case scenario, say an error in the cooling system of a nuclear power plant, may be too expensive for us to afford.

Much more rational is the physicist approach. Physicists, first build models, called theories, of the world. They measure the plausibility of these theories by the degree of falsifiability, i.e. by exposing them to experiments, *testing* them for flaws against the real world. This is quite similar to the state of the art in industrial software quality management, which is extensive testing. The real world corresponds to the *implementation* of the running computer system, the theories are called *formal specifications*, and test-vectors are merely more than experiments. Even though this approach is much more justified to be trusted than the traditionalist one, on the long run it is futile. With a complexity of programs increasing in an exponential way, it is almost impossible to ensure that *no* bugs are left undetected. Take for example operating systems: within twenty years the program size increased by several orders of magnitude, from a few hundred kilobytes (MS DOS) to over a Gigabyte (Windows 7). As physicists never claim the correctness of their theories, testing cannot prove the absence of errors, either.

Proofs, yet, lie in the domain of mathematicians. They not only claim correctness of theorems. But they have developed precise rules, called axioms, to formalize and to derive the validity of such claims. Correspondingly, one could axiomatize software systems and deduce correctness properties. Still, the mathematicians' approach suffers from a serious problem: Who ensures that the proof has been correctly deduced? What is the worth of a 'proof' written down on hundred of pages, whose formalism is only understood by a handful experts? The answer is probably: not much, as the following example illustrates. An important result in topology, the jordan curve theorem, was found by Camille Jordan. He stated in his publications in 1887 '*that this theorem is clearly true*' and provided a proof which later on was dismissed over decades as insufficient or simply as wrong. In 1905 Oswald Veblen claimed a new proof, which however used a series of 'intuitive' arguments making it hard do determine its truth value. The solution was reached in 2005 when a *computer checked* proof was conducted by Hales [Hal07] (he also found out that Jordan's original proof did not contain any serious flaw).

This, finally, may be also the silver bullet for trusting computer systems: *formal verification*. Formal verification, the computer scientists' approach, comprises components of all three mentioned ways: *implementations* of computer systems are axiomatized and verified against *formal specifications*. These *proofs* are then *computer checked*, giving us the ultimate amount of confidence. Nevertheless, the specification still has either to be *tested* or simply to be *trusted*.

Pervasive System Verification as a Scientific Task This approach was pioneered by, among others, Dijkstra, Flyod, Lamport, and Hoare. Up to now most approaches in formal verification only model some limited parts or high-level layers of an overall system. Underlying layers are either modeled in some semi-formal way or some *perfect* (i.e. correct) and simplified model is even postulated, claiming an accurate model would be too complex to build. For example when a concrete application, as a fingerprint authentication software, running on top of a real operating system, as *Linux*, should be verified, it is not enough only to prove the correctness of the abstract fingerprint matching algorithm. This is because the specification of this algorithm may use some model of the services provided by the operating system, *assuming* both correctness of modeling and of functionality of the services. Moreover, only very few approaches carry out the verification at the level of *concrete code*.

Now, would the goal to *pervasively* formalize and verify the correctness of a computer system, from transistors to software not be a major scientific task? Having in mind that proving comes closest to understanding, would it not be worthwhile to entirely *understand* one of the probably most complex creations of men? In [Moo03] J. S. Moore, principal researcher of the CLI stack project [BHMY89], gives a strongly affirmative answer to this question and declares the formal verification of a '*practical computing system*' as a grand challenge problem.

A main goal of the Verisoft project [The09] is to tackle this challenge. In the academic system, a subproject of the Verisoft project, a general-purpose computer system, covering all layers from the gate-level hardware description to communicating concurrent programs should be designed, implemented and verified. The verification is supposed to be pervasive throughout all layers of abstraction and all models and proofs should be formalized and checked in a single mechanized language, Isabelle/HOL [NPW02].

Motivation for the thesis Almost a decade prior to the Verisoft project the well-perceived CLI stack project [BHMY89] tackled the challenge of pervasive system verification — but left many issues open. Even though, impressive progress had been accomplished, several crucial issues have not been treated. Among those, most prominently, the integration and verification of device drivers and of high level applications, as client/server software, were omitted. This thesis tackles these two open challenges within Verisoft:

- **Formal device driver verification.** Device drivers are an integral part of system software. Not only high-level functionality such as file I/O or networking depend on devices. Even basic operating system features, such as demand paging, need correctly implemented drivers. Hence, any verification approach of computer system stacks should deal with driver correctness. Nevertheless, when proving functional driver correctness it does not suffice to reason only about code running on a processor. Devices themselves and their interaction with the processor also have to be formalized. On the lower-level this results in a computational model, in which the devices and the processor take turns in execution. Even in this concurrent context, the verification can be kept largely sequential and modular with respect to the other devices. This is made possible by sound reordering of computation traces, given that devices do not interfere with each other and the driver monopolizes a single device.

In this thesis, we show how to pervasively and formally integrate devices and their drivers into a language stack reaching from the level of assembly up to the level of high-level languages. This stack is leveraged for the formal verification of a simple hard disk driver, which is subsequently embedded into the correctness of the micro kernel used in Verisoft. To the best of our knowledge, this marks the first formal functional verification of a device driver against a realistic device and system model. The extension of the language stack, the verification of the hard disk driver (write function) and its embedding into kernel correctness have been accomplished formally in the computer-aided verification system Isabelle/HOL.

The results reported in this context are based on the following publications. The overall verification approach in the Verisoft project is described in [AHL$^+$08]. The extension of the language stack to reason about devices can be found in [AHL$^+$09, AHL$^+$08]. The assembly verification of the hard disk driver and a small reordering theory for the concurrent system was published in [AH08]. In [ASS08] we report on its integration in the kernel. The specification of other device models, as the serial interface can be found in [AHK$^+$07].

- **Proving the Correctness of Client Server Software.** For a long time programmers struggled with the time consuming implementation of communication protocols for distributed systems. In 1984 Birell and Nelson [BN84] responded to this problem and proposed the first Remote Procedure Call (RPC) mechanism. They suggested to allow programs to call procedures located on other machines. When a process on machine A (client) calls a procedure on machine B (server), the calling process on A is suspended, and execution of the called procedure takes place on B. The caller eventually regains control, extracts the results of the procedure, and continues execution. Information can be transported from the caller to the callee in the parameters and return in the procedure result. Neither message passing nor I/O is visible to the programmer. RPC became a widely-used technique that underlies many distributed operating systems [TR85].

  Thus, lying at the heart of any client server software, the formal specification and verification of RPC mechanisms is a prerequisite for the verification of any such software. In this thesis, we present a mathematical specification of an RPC mechanism based on the formalization of the Simple Operating System (SOS) [Bog08]. Furthermore, we outline how to prove the correctness of an implementation of this mechanism at the code level. To the best of our knowledge this is the first treatment of the correctness of an entire RPC mechanism at the code level.

The results on client/server correctness have been published in [ABP09].

The most challenging aspect of this work is to pervasively integrate a wide range of models, theories, and proofs — work of many researchers at different locations — to finally obtain correctness theorems which target the whole computer stack. This has been accomplished formally in Isabelle/HOL for the device driver part and on paper for client/server correctness. Major parts of the formal theories of Verisoft have been published or are in the process of publication [HP08].

1.1 Structure of the Document

The thesis is structured as follows. In the remainder of this chapter we first give an overview on the related work; then we sketch the system and language stack developed in Verisoft and point out our contribution therein; and finally we summarize the mathematical notation used in this work. Chapter 2 is designed as an road map for the theories and theorems given in the thesis. Part I of the thesis deals with functional formal driver verification:

- In Chapter 3, we develop methodology and extend the current Verisoft stack to embed reasoning about device drivers.
- In Chapter 4, this stack is leveraged to verify a simple hard disk driver and integrate it into overall kernel verification.

Part II of the thesis shows how correctness of client/server software is proven.

- In Chapter 5, the underlying formal model of the simple operation system is introduced.
- In Chapter 6, upon this model we specify an RPC mechanism and prove the correctness of a simple math server.

Finally, in Chapter 7 we outline possible future work and conclude.

1.2 Related Work

In the following, we discuss related work within each of the four areas, which are: system software verification, reordering theory, device modeling and driver verification, as well as RPC.

System stack verification First attempts to use theorem provers to specify and even prove correct operating systems were made as early as the seventies in PSOS [NF03] and UCLA Secure Unix [WKP80]. However a missing or to a large extend underdeveloped tool environment made mechanized verification futile. With the CLI stack [BHMY89], a new pioneering approach for pervasive system verification was undertaken. Most notably the simple kernel KIT was developed and its machine code implementation was proven correct. Compared to modern kernels KIT was very limited, in particular it lacked the interaction with devices. The project L4.verified [HEK+07a] focuses on the verification of an efficient microkernel, rather than on formal pervasiveness, as no compiler correctness or an accurate device interaction is considered. The microkernel is implemented in a substantial subset of C, including pointer arithmetic and an explicit low-level memory model [TKN07]. However with inline assembly code we gain in Verisoft an even more expressive semantics as machine registers become visible if necessary. So far only exemplary portions of kernel code were reported to be verified, the virtual memory subsystem uses no demand paging [TK04]. For code verification the L4.verified project relies on Verisoft's Hoare logic environment [Sch06].

In the FLINT project, an assembly code verification framework is developed and code for context switching on a x86 architecture is formally proven correct [NYS07]. Although a verification logic for assembler code is presented, no integration of results into high-level programming languages has been reported yet. The VFiasco project [HTS02] aims at the verification of the microkernel Fiasco implemented in a subset of C++. Code verification is performed in an embedding of C++ in PVS and there is no attempt to map the results down to the machine level.

Moreover, we refer the interested reader to [Kle09], in which Klein provides an excellent and comprehensive overview of the history and current state of the art in operating systems verification.

Reordering theory Reordering is used in this thesis in two different flavors: for reasoning about the concurrent execution of a processor and devices, and for determining non-interference properties on operating system calls.

The reordering of execution sequences to obtain atomic specifications was studied in literature under the topic of *reduction theorems*. Lipton proved safety properties of pre-/ post-condition style sequentially and propagated these to the implementation [Lip75]. Cohen and Lamport extended this to liveness and a more fine-grained analysis of the reordered parts of the sequence [CL98, Coh00]. Most reduction theorems assume that the implementation fulfills some non interference theorems. In contrast we prove this assumption on the atomic specification by exploiting a similar insight as reported in [SC06]. Justified by the memory mapped I/O architecture and the SOS semantics respectively, the theories presented here are a specialization, enabling us to formulate even stronger reduction theorems than reported in the literature.

Drivers So far most device related verifications have either targeted the correctness of gate-level implementations or safety properties of drivers. In approaches of the former kind, simulation and test based techniques are used to check for errors in the hardware designs. In particular, [BKS03, RPS01] deal with serial interfaces in that manner. In approaches of the latter kind the driver code is usually shown to guarantee certain API constraints of the operating system and hence cannot cause system crashes. For example, the SLAM project [BR01] provides tools for the validation of safety properties of drivers written in C. SLAM's success led to the deployment of the Static Driver Verifier (SDV) as part of the Windows Driver Foundation [Mic04]. SDV automatically checks 65 safety rules concerning the Windows Driver API for device drivers. Hallgren *et al.* [HJLT05] modeled device interfaces for a simple operating system written in Haskell. Three memory-mapped I/O calls were specified: read, write, and test for valid region. However, the only correctness property being stated is the disjointness of the device address spaces.

In contrast to all mentioned approaches, we aim at the formalization and *functional* verification of drivers interacting with a device. Thus, it is not sufficient to argue about the device or programming model alone. Even in other ongoing systems verification projects, the L4.verified project [HEK+07b] and the FLINT project [FLI], device behavior and driver correctness are not considered. To our knowledge, the only work similar in scope is the challenge proposed by Holzmann [Hol06] dealing with the formal verification of a file system for a Flash device. In response to the challenge, Woodcock reports on the partial specification of the file system (the 'file store') and a refinement proof mapping the store to a Java program [FFW07]. Simultaneously, the Flash hardware is being formalized [BW07]. Verifying a low-level Flash driver and integrating it into the filesystem proofs are future work. Concurrency is not an issue since only a single device is considered.

Two earlier Verisoft publications are relevant for our work. In [HIP05] paper-and-pencil models and proofs related to a simple disk driver are reported. We extend this work in three important ways: models and proofs are formalized in Isabelle/HOL, and the models are now concurrent instead of lock-step. Thus, they are not restricted to disks, which are 'simple' for lack of external communication. Moreover, in this thesis we formally integrate the proofs into the higher language stack. In [AHK+07] we have reported on formal models of a serial interface and an architecture with devices, but not treated drivers.

Client/Server A well-known implementation of RPC was provided by Sun Microsystems [Sri95]. The best example for an application using RPC is probably the network file system (NFS) [SCR+03].

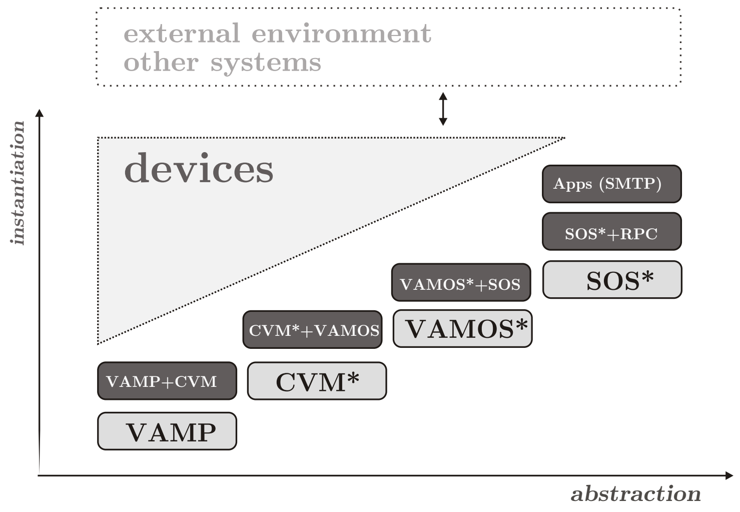


Figure 1.1: Verification stairs of the academic system (Verisoft) — Abstract models with star, implementation models without

The challenge of verifying a simple RPC memory system proposed 1994 by Broy and Lamport triggered a widespread response. The result was impressive: as many as 15 different solutions were published [BMS96]. Their goal, however, was to compare different formalization techniques and proof methods on an abstract case-study for distributed computing — rather than providing programmers with a verified RPC mechanism. In contrast, we are aiming at demonstrating that verification of RPC in a real setting, running under a real operating system is feasible. Our specifications have been used to specify and partially prove properties of user applications [LNRS07, BBBW07].

1.3 The Setting

System Stack. Our system stack comprises many layers. A model of one layer is derived from the model of the (next) lower layer by means of instantiation and abstraction (cf. Figure 1.1). The *hardware architecture* is called VAMP [Tve09], a DLX like processor that supports address translation and memory-mapped I/O devices. With the next level of *communicating virtual machines* (CVM) a hardware-independent programming interface for a microkernel is provided [IT08, Tsy09]. This establishes the notion of separate concurrent user processes with virtual memories. Memory virtualization is implemented by means of demand paging [Sta09], which in turn relies for page swapping on the invocation of a hard disk driver. Parts of the CVM (as the hard disk driver) are implemented in assembly, because C0, our main im-

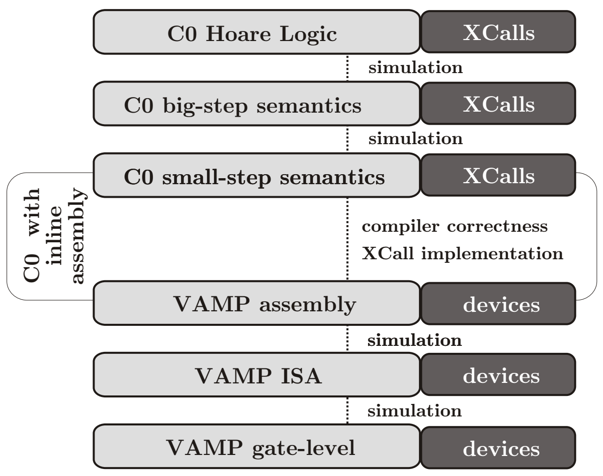


Figure 1.2: Semantics stack

plementation language and a subset of C, lacks some low-level programming constructs. On the basis of the CVM our *microkernel* VAMOS [DDB08] is programmed in pure C0 (with external functions provided by the CVM). The *simple operating system* (SOS) is implemented as a (privileged) user process of VAMOS [Bog08]. It offers file I/O, network access and inter-process communication. On top of it user applications are provided with a client/server architecture based on remote procedure calls [ABP09]. Finally these user applications implement the functionality of the academic system: signing software, SMTP client and server [LNRS07], and a simple email user agent [BHW06]. The implementation stack is also depicted in Figure 1.1.

The contribution of this thesis to the system stack is twofold. First, we formally verify and embed the correctness of a hard disk driver into the overall verification of the CVM. Moreover, in the course of the formal integration, we met a series of unforeseen difficulties, as for example the necessity to reason about memory consumptions of compiled code. We solve this problems, by developing and verifying general methodology. Second, on top of the SOS we deliver a new model, which supports RPC primitives and allows to implement and verify client/server applications.

Next, we introduce the semantics stack which is orthogonal to the system stack described before. With the semantics stack a convenient Hoare logic to reason about the sequential parts of C0 programs (without inline assembly code) is established. We extend this stack to provide means to deal in Hoare logic with assembly code and to integrate devices.

Semantics Stack. The C0 semantics stack comprises a Hoare logic, a big-step semantics, and a small-step semantics, and can be continued to the VAMP machine level, which is divided further into assembly layer, instruction set architecture, and gate-level hardware. An overview is depicted in Figure 1.2. By a higher level of abstraction in the Hoare logic compared to the small-step semantics, efficiency for the verification of individual C0 programs is gained. However, since the semantics stack is merely a proof tool for C0 programs, the results obtained in the Hoare logic have to integrate into our systems stack. The stack supplies soundness and simulation theorems that permit the transfer of program properties from the Hoare logic down to the small-step semantics. Those properties can be mapped to assembly machines by applying compiler correctness. We can get further down to the ISA layer by employing a simulation theorem and finally to the hardware by employing a processor correctness result.

The Hoare logic provides sufficient means to reason about pre and post-conditions of sequential, type-safe, and assembly-free C0 programs. Compiler correctness, though, is formulated at the small-step semantics level. This allows the integration with inline assembly code or concurrent computations, e.g. introduced by devices. The big-step semantics is a bridging layer, which is convenient to express the results of the Hoare logic operationally. The differences reflect the purpose of the layers. The Hoare logic is tuned to support verification of individual programs, whereas the small-step semantics is nearer to the architecture level.

Up to now we have argued how to bring the results down to the lower levels such that we can conduct reasoning at a comfortable abstraction level. However, this comes at the cost of expressiveness, as the lower levels not only introduce 'unnecessary clutter'. Most prominently, the levels below C0 allow the integration of devices, which are a concurrent source of computation. As soon as we attempt to reason about C0 programs that use these devices we either have to be able to express device operations at the Hoare logic level or we are doomed to carry out the whole verification at the assembly level. Our approach is to abstract the effect of those low-level computations into atomic *XCalls* (extended calls) in all our semantic layers. The state space of C0 is augmented with an additional component that represents the state of the external component, e.g. the device. An XCall is a procedure call that makes a transition on this external state and communicates with C0 via parameter passing and return values. With this model it is straightforward to integrate XCalls into the semantics and into Hoare logic reasoning. The XCall is typically implemented in assembly. An implementation proof of this piece of assembly justifies the abstraction to an atomic XCall.

Somewhat similar to the XCalls in the C0 semantics layers [Sch06], devices are added to all the semantic layers of the VAMP. Their state and transition functions are shared between all layers. These transition functions as well as the VAMP semantics describe small-step computations, which are interleaved to obtain the concurrent computation of the combined system. One central prerequisite to employ our individual transfer and correctness results to obtain a global property for the combined system is to disentangle the different computations by means of reordering.

This thesis contributes to the semantics stack by integrating devices at the assembly level and by introducing the concept of XCalls to reason about low-level entities in higher level languages. By this we develop a pervasive methodology to formally verify functional correctness of device drivers and reason about them in Hoare logic. This is in particular challenging, since it integrates the concurrent computations of the processor and devices at the architecture level into the sequential view provided by the C0 (and assembly) language.

1.4 Notation

Basic types, terms and operations The basic sets used in this document are $\mathbb{N}$ (naturals including zero), $\mathbb{Z}$ (integers), $\mathit{bool} = \{\mathsf{true}, \mathsf{false}\}$ (booleans), $\mathbb{B}$ (binary digits) and *string* (strings). The sets type_n, fun_n, var_n, field_n denote names and are all assumed to be subsets of *string*. The power set of A is denoted by $\mathit{pow}(A)$.

For the numeric types we take the operations: $-$ (subtraction), $+$ (addition), $\div$ (integer division), $\cdot$ (multiplication), $\bmod$ (modulo), and $\sum$ (sum) for granted. With $a \;\mathsf{dvd}\; b$ we express that b is dividable by a. For an operation $\mathsf{op} \in \{+, -, \cdot\}$ we additionally define corresponding operations modulo 32 as follows: $a \;\mathsf{op}_{32}\; b = (a \;\mathsf{op}\; b) \bmod 2^{32}$.

For boolean expressions we assume that the basic logical operations: $\wedge$ (conjunction), $\vee$ (disjunction), and $\neg$ (negation) are defined. We write $\exists$ for the existential quantifier and $\forall$ for the universal quantifier. The type of a quantified variable is omitted, if it can be inferred otherwise. Moreover, we may write $p(\dots)$ instead of $\forall x \,.\, p(x)$, and $p(?)$ instead of $\exists x \,.\, p(x)$. An implication is denoted by $\Longrightarrow$ and equivalence by $\equiv$. Moreover, we use inference rules of the form $\dfrac{a \quad b}{c}$ to denote implications $a \wedge b \implies c$.

For terms, besides the ordinary **if** $-$ **then** $-$ **else** expression we use abbreviations. We write **let** $x_0 = y_0; ...; x_n = y_n$ **in** $e(x_0, .., x_n)$ as an abbreviation for $e(y_0, .., y_n)$. Note that an assignment in a **let** expression may be nontrivial. This is because pattern matching can be used. We write, for example, **let** $(x_0, x_1) = (y_0, y_1)$ **in** $x_0 + x_1$ to simultaneously assign abbreviations for x_0 and x_1, which are later used separately.

Binary Operations The set of bitstrings of length n is denoted by $\mathbb{B}^n$. Given a bitstring $a = a_0 \dots a_n$, we denote with a_i the ith bit of the string. The common bitwise logic operations $\wedge_n$ (bitwise and), $\vee_n$ (bitwise or), $\otimes_n$

(bitwise exclusive or) and shift operations $<<^l_n$ (logical left shift), $>>^l_n$ (logical right shift) and $>>^a_n$ (arithmetic left shift) are taken for granted.

We interpret bitstrings either as (unsigned) binary or (signed) two's complement numbers. The bitstring a denotes the binary number $\langle a\rangle_n = \sum_{i=0}^{n-1} a_i \cdot 2^i$ and the two's complement number $[a]_n = -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i$. With bitstrings of length n one could encode binary numbers in the set $\{0, \ldots, 2^n-1\}$ and two's complement numbers in the set $\{-2^{n-1}, \ldots, 2^{n-1}-1\}$. On the counterpart the functions *to-bin* and *to-two* denote the translations from numbers to bitstrings.

We expand operations on bitstrings to integers as follows. Given the two integers a and b, then the operations op^t with $\mathsf{op} \in \{\vee_n, \otimes_n, <<^l_n, >>^l_n, >>^a_n, +_n, -_n\}$ are defined by $a \; \mathsf{op}^t \; b \equiv [(\mathit{to\text{-}two}(a) \; \mathsf{op} \; \mathit{to\text{-}two}(b))]_n \mod 2^n$.

Functions and sequences We use common lambda notation to define functions, for example for addition we may write $\lambda x\, y \,.\, x + y$. Functions are always assumed to be total. Any function may, however, return with the special value *undef*, denoting that the function is undefined at this position. We denote with the *range* of a function f all possible outputs, i.e $\mathit{range}(f) \equiv \{y | y \neq \mathit{undef} \wedge (\exists x.f(x) = y)\}$. Analogously, the *domain* of a function is defined by $\mathit{dom}(f) \equiv \{x | f(x) \neq \mathit{undef}\}$. Updates to a function f at position y to value v are abbreviated by $f(y := v) \equiv \lambda x \,.\, \textbf{if } x = y \textbf{ then } v \textbf{ else } f(x)$.

Functions with domain $\mathbb{N}$ are called sequences. We say a predicate Q is live in some sequence *seq* if Q is infinitely often true: $\mathit{live}(\mathit{seq}, Q) = \forall i \,.\, \exists j > i \,.Q(\mathit{seq}(j))$. Next, we introduce filters on sequences. A filter is a function which takes a sequence and a predicate as input. It returns a new sequence in which all elements are filtered out for which the predicate does not hold. We first define the function *next* which returns for a given sequence *seq*, predicate Q and position number k, the smallest position number $l > k$ in the sequence *seq* at which Q holds.

$$\mathit{next}(\mathit{seq}, Q, k) = \begin{cases} l & \textbf{if } l > k \wedge Q(\mathit{seq}(l)) \wedge (\forall i < l.i > k \implies \neg Q(\mathit{seq}(i))) \\ \mathit{undef} & \textbf{otherwise} \end{cases}$$

We define the filter now, by the following helper function filter_h:

$$\begin{array}{lll} \mathit{filter}_h \; (\mathit{seq}, Q, 0) & = & \textbf{if } Q(\mathit{seq}(0)) \textbf{ then } 0 \textbf{ else } \mathit{next}(\mathit{seq}, Q, 0) \\ \quad\;\; (\mathit{filter}_h, Q, n+1) & = & \mathit{next}(\mathit{seq}, Q, \mathit{filter}_h(\mathit{seq}, Q, n)) \end{array}$$

$$\mathit{filter}(\mathit{seq}, Q) = \lambda n.\mathit{seq}(\mathit{filter}_h(\mathit{seq}, Q, n))$$

Obviously, applications of the filter function are not always well-defined. This holds for example if the given predicate is never fulfilled. The following lemma shows that liveness is a criteria for well-defined filters:

Lemma 1 (Well-defined filters)

$$\mathit{live}(\mathit{seq}, Q) \implies \forall i.\mathit{next}(\mathit{seq}, Q, i) \neq \mathit{undef}$$

Tuples Let $T_1, .., T_n$ be types. Then the tuple type $T_1 \times .. \times T_n$ yields the set of all elements $(v_1, .., v_n)$ with $v_1 \in T_1, .., v_n \in T_n$. Tuples of size two (i.e. for $n = 2$) are called pairs. Given a pair (a,b), then we define the two access operators first and second as follows: $\mathit{fst}((a, b)) = a$ and $\mathit{snd}((a, b)) = b$.

Records Let $T_1, .., T_n$ be types, $\mathsf{fn}_1, .., \mathsf{fn}_n$ be strings. Then the record type $(\mathsf{fn}_1 : T_1, .., \mathsf{fn}_n : T_n)$ yields the set of all elements $(\mathsf{fn}_1 = v_1, .., \mathsf{fn}_n = v_n)$ with $v_1 \in T_1, .., v_n \in T_n$. Let r be a record of type $R = (\mathsf{fn}_1 : T_1, .., \mathsf{fn}_n : T_n)$. We denote the access to the record field fn_i with $r.\mathsf{fn}_i$. An Update to the field fn_i by a value v is abbreviated by $r[\mathsf{fn}_i := v]$. It yields the new record $(\mathsf{fn}_1 = r.\mathsf{fn}_1, .., \mathsf{fn}_i = v, .., \mathsf{fn}_n = r.\mathsf{fn}_n)$ if $i \in \{0, .., n\}$ and is undefined otherwise.

Abstract data types Let $T_{1.1}, .., T_{1.m_1}, .., T_{n.1}, .., T_{n.m_n}$ be types and $\mathsf{C}_1, .., \mathsf{C}_n$ be constructor names. We declare an abstract datatype T as:

$$\begin{array}{lll} T ::= & \mathsf{C}_1 \textbf{ of } T_{1.1}, .., T_{1.m_1} & \mid \\ & \ldots & \mid \\ & \mathsf{C}_n \textbf{ of } T_{n.1}, .., T_{n.m_n} & \end{array}$$

If, for example $(x_{1.1}, \ldots, x_{1.m_1}) \in T_{1.1} \times .. \times T_{1.m_1}$ then $\mathsf{C}_1(x_{1.1}, \ldots, x_{1.m_1})$ is an element of the above defined abstract data type. Note, that abstract data types may be defined recursively. Pattern matching may be used on the abstract data type either in function definitions, or by the **case-of** construct.

Sometimes we use parametrized types, which are functions taking a type and constructing a new one. Usually we denote such type functions by $\alpha\ T$, where α is the type parameter.

An example of a parametrized abstract data type is the option type. It has two constructors, NONE and SOME:

$$\begin{array}{ll} \alpha\ \mathit{option} ::= & \mathsf{NONE} \mid \\ & \mathsf{SOME} \textbf{ of } \alpha \end{array}$$

For convenience we abbreviate NONE with $\perp$ and $\mathsf{SOME}(x)$ with $\lfloor x \rfloor$. We define the selection operator the by $\mathsf{the}(\lfloor x \rfloor) = x$ and $\mathsf{the}(\perp) = \mathit{undef}$. Option types are often used to specify the output of computations. Then, $\perp$ stands for a computation that is stuck.

Lists We define a list type over elements of type α as a recursive data type, with two constructors, one for the empty list and one for concatenation of a single element:

$$\begin{array}{ll} \alpha\ \mathit{list} ::= & \mathsf{NIL} \mid \\ & \mathsf{CON} \textbf{ of } \alpha \times \alpha\ \mathit{list} \end{array}$$

For convenience we introduce the abbreviations $[]$ for the empty list NIL and $x\#xs$ for the concatenation $\mathsf{CON}(x, xs)$. The functions hd and tl return the head and the tail of a list, respectively, i.e. it holds $hd(x\#xs) = x$ and $tl(x\#xs) = xs$. Moreover, we abbreviate concatenation of many elements, as for example of $a\#(b\#(c\#[]))$ by writing $[a, b, c]$. We denote the the size of a list xs by $length(xs)$. Access to list element i is denoted by $xs!i$, where the head element is $xs!0 = hd(xs)$. Updates of the ith list element to some value v is denoted by $xs[i := v]$.

Given a list of pairs, we may interpret the first element of each pair as a key and the second one as the data. We define a lookup for a given key id by $xs?id \equiv snd(xs!(\min\{i \mid fst(xs!i) = id \land i < length(xs)\}))$. Concatenation of two lists of the same type is denoted by $xs@ys$. Finally, we define an operator which returns for a given list a set containing all its elements. It is defined by $set(xs) \equiv \{x \mid \exists i < length(xs).\ xs!i = x\}$.

Chapter 2

Road Map

This thesis is composed of two parts: in the first part we report on the formal functional verification of device drivers, while in the second part we elaborate on the correctness of client/server software. The remainder of this chapter may be read as a manual to the thesis or as its summary, alternatively.

2.1 Formal Verification of Device Drivers

Part I is divided into two portions: First, we develop the extension of the language stack, which covers device semantics and enables the verification of drivers in the setting of traditional program logic (Chapter 3). Then, we leverage this stack to verify a hard disk driver and integrate the results into the overall kernel correctness (Chapter 4).

Our journey starts in Section 3.1 with the aim of defining and justifying a low-level programming model for device drivers running on the VAMP processor. Therefore, we first elaborate on the three levels of VAMP abstraction: VAMP gate-level, VAMP ISA and VAMP assembly. Subsequently, we introduce a general device model, and show how devices are embedded at each of the VAMP levels. At gate-level we encounter a lock-step model, in which all devices and the processor execute in parallel, whereas in the upper models granularity on time is lost. This loss is modeled by a non-deterministic oracle and interleaved semantics. Adjacent models of the VAMP are linked to each other via simulation theorems, which are used to propagate properties proven on abstract levels to lower levels. Most importantly, in this thesis, we are concerned with simulation between VAMP ISA with devices and VAMP assembly with devices and in the resulting software restrictions (Theorem 1).

Although we are arguing about a concurrent system, we want to use conventional verification technology — say verification condition generators for Hoare logic — to argue about (as large as possible) sequential portions of the interleaved computation. Therefore, in Section 3.2 we introduce a small reordering theory, which (i) provides us with theorems to embed reasoning about the interleaved model into sequential programs and (ii) which simplifies to verify low-level drivers. We state several lemmas and theorems used to separate verification of processor and device computations (Theorems 2 and 3).

In Section 3.3 we describe the C0 small step semantics and outline the compiler correctness theorem (Theorem 4), which links C0 computations to the computations of the compiled code running on VAMP assembly. We proceed with extending compiler correctness to a target machine with device computations (Theorem 5). Basically, correctness follows from the fact that the compiled code never accesses devices and from the application of the reordering theory. Complier correctness imposes a set of conditions on the memory consumption of the target machine. Such conditions are of dynamic nature, i.e. they have to be discharged at each step of the computation. Hence, we verify a method to statically estimate the memory consumption of C0 code (Theorem 6).

In Section 3.4 we show how inline assembly (with no device access) can be embedded into C0 code, define an appropriate new semantics and state its soundness against VAMP assembly (Theorem 7). Subsequently, we extend C0 with inline assembly to allow device access. The new transition system enables us to reason completely sequentially on C0 code and separate the verification of assembly drivers for different devices. The soundness of this model against VAMP assembly is based on the reordering theory developed before (Theorem 8).

To capture the semantical effects of drivers and embed them into Hoare-style reasoning, we introduce the concept of XCalls. XCalls are basically atomic state updates on the C0 machine and an arbitrary state extension, which is used to abstract from low-level entities (as e.g. devices). In Section 3.5 we formulate an adequate semantics of C0 with XCalls, define the notion of implementation correctness of an abstract call, and prove an extended compiler correctness theorem (Theorem 9). The proof is conducted by relying on the semantics for C0 with inline assembly accessing devices.

In Chapter 4 the obtained new and extended language stack is applied to verify a hard disk driver and embed the result into the verification process of the CVM kernel. We start in Section 4.1 by defining a formal model of the hard disk. Then, in Section 4.2 the assembly driver for writing memory pages is described and its correctness stated (Theorem 10). Verification is carried out by applying the reorder theory presented beforehand. Next, in Section 4.3 we embed the assembly driver into C0, specify it in terms of the XCall semantics and prove its correctness (Theorem 11). The verification is carried out by applying the XCall theory developed previously. Finally, we outline how this theorem is used in the context of the page-fault handler verification.

2.2 Verification of Client/Server Software

Part II is divided into two portions: First, we describe the formal model upon which the client/server software is implemented and introduce a verification methodology to deal with non-interfering system calls (Chapter 5). Then, we leverage this model and technique to specify the correctness of an RPC implementation and verify a simple math server (Chapter 6).

We start in Section 5.1 with a brief overview of the functionality of the simple operating system and by outlining its position in the Verisoft system stack. In Section 5.2, we specify a subset of Bogan's SOS*, a distributed model of computation, describing the interactions of the simple operating system SOS with its user processes. Thereby, user processes are modeled in a quite abstract way such that they can be instantiated to model user programs in different programming languages. To be realistic, the model necessarily has to include inter process communication (IPC) and basic port mapper functionality. Furthermore, to be able to derive any useful termination properties, fairness of the scheduler has to be postulated on the model. In the SOS* model, timeouts are restricted to finite and infinite timeouts.

Similar to the concurrent model of processor and devices, we deal with a distributed system. Desirably, here again, conventional programming logic should be used to argue about sequential portions of the distributed computation. For that purpose, in Section 5.3, we develop (similar to the reordering theory) a simple theory of non interfering system calls (Theorem 12).

In Section 6.1, we instantiate the user process model to C0 applications. However, pure C0 small step semantics is not sufficient to describe these applications: semantics of system call invocations have also to be formalized. We outline how to extend such small steps semantics for system calls like the portmapper- and IPC calls. This permits us to show that a service can indeed be looked up after it has been registered at the portmapper (Lemma 39). It seems very attractive to apply the concept of XCalls developed previously also to specify system calls. However, in contrast to the device model, our goal here, is not to hide all the interleaving in a single atomic step. This is not possible, since the concurrent interaction of the operating system and the processes should remain visible.

In Section 6.2, we show how to specify signatures of services in terms of C0 types.

Interface compilers generate from such signatures a library of external C0 functions implementing the remote procedure call primitives `RPCsend` and `RPCrecv`. The implementation of these functions uses the (previously defined) IPC primitives. One has to be able to show termination, if timeouts are finite (Lemma 40). Furthermore, one has to show that matching `RPCsend` and `RPCrecv` calls either lead to a timeout or to a rendezvous situation and that, in the latter case, the data is correctly communicated (Lemma 41).

The RPC primitives `RPCsend` and `RPCrecv` as well as the portmapper calls suffice to implement a wide variety of protocols. We demonstrate this in Section 6.5 for a simple example of an RPC client protocol.

A library, providing functions implementing the complete client protocol for calling remote procedures, can be generated by a compilation process for interfaces. For the functions generated by this library, one has to be able to show that they look up services and, in the case of success, eventually send calls to the server and receive answers (Theorem 13). Theorem 13, just like Lemma 41, is carefully phrased to reflect the fact that protocols may get stuck if some clients do not obey the required protocol.

Finally, in Section 6.6, we implement, with the given primitives, a simple Math Server. This server registers its services at the portmapper and then services remote procedure calls. If all clients adhere to the protocol, then one can show that the protocol never gets stuck and that remote procedure calls with infinite timeouts eventually get answers (Theorem 14).

Part I

Formal Function Verification of Device Drivers

Chapter 3

Extending the Language Stack

3.1 VAMP and Devices

The goal of this section is to develop a justified, low-level programming model for drivers.

The processor of choice in Verisoft is called ***V**erified **A**rchitecture **M**icro **P**rocessor* (VAMP). It has a non trivial design, supporting fix-point arithmetic, out-of-order execution, internal and external interrupts and two different execution modes: system and user mode. In the latter, memory access is subject to address translation. Devices are integrated to the VAMP by means of memory-mapped I/O.

We model the VAMP at three different levels of abstraction: (i) VAMP gate-level, which describes the gate-level implementation, (ii) VAMP ISA which is the corresponding instruction set architecture, and hence the specification model to VAMP gate-level, (iii) and VAMP assembly, which is intended to be a convenient layer for implementation and verification of low-level applications. It abstracts from certain aspects of lower layers which are irrelevant for most applications.

At each level, devices are modeled as finite state automata communicating with the processor and a not modeled external environment. However, the interaction model between the VAMP and the devices differs from level to level. At gate-level the processor and the devices are executed in lock-step, i.e. in each clock cycle each device and the processor take a transition. At VAMP assembly and VAMP ISA, devices and the processor are executed interleaved.

We start by elaborating on the computational models of the VAMP without devices. Subsequently, we introduce the general device model and show how devices are embedded. By establishing two simulation theorems between the adjacent models, properties shown for VAMP assembly with devices can be transfered to VAMP ISA with devices and finally to the gate-level implementation of the hardware. We conclude this section by outlining and discussing both simulations.

Note, that the models are introduced bottom-up, from a coarse overview of the gate-level implementation to a fine-grained assembly description, which will serve as the target language for driver verification.

The VAMP architecture is based on the DLX architecture [HP96] and was initially presented in [MP00]. An implementation of the VAMP has been formally verified in 2003 [BJK+03, BJK+06]. Since then, the VAMP has been extended with address translation and support for I/O devices [AHK+07, DHP05, TS08]. Details on the gate-level implementation of the VAMP and its verification can be found elsewhere [Tve09, TA08].

3.1.1 Gate-level Model

A configuration $h \in C_h$ of the VAMP gate-level model consists of

- 32 bit registers, which are visible for the programmer. These are the program counters, the general purpose registers and the special purpose registers,
- additional 32 bit implementation specific registers, which are invisible for the programmer,
- a random access memory (RAM).

The VAMP is a pipelined machine with many stages supporting forwarding and out-of-order execution. For the embedding of devices two stages are of special interest (see Section 3.1.4): (i) the memory stage, in which store and write operations access the memory and (ii) the write-back stage, in which instructions are written back and interrupts become visible.

The transition function of the hardware takes as input the current configuration of the hardware and an external event vector representing interrupt signals of the connected devices. It returns the updated hardware configuration after one cycle.

3.1.2 Instruction Set Architecture

The instruction set architecture VAMP ISA fulfills two aims: (i) *downwards* it represents the specification for the gate-level hardware design, (ii) *upwards* it provides the system programmer with the model describing all visible components and behavior, including address translation and interrupt handling. In this section we only give an overview over the transition system. A full-blown description of VAMP ISA can be found in [MP00].

State Space

Processor configurations are of the record type C_{isa}, which consists of the following fields:

$\mathsf{pc} \in \mathbb{B}^{32}$	The program counter.
$\mathsf{dpc} \in \mathbb{B}^{32}$	The delayed program counter (used to specify the delayed branch mechanism detailed in [MP00]).
$\mathsf{gpr} :: \mathbb{B}^5 \to \mathbb{B}^{32}$	The general purpose register file.
$\mathsf{spr} :: \mathbb{B}^5 \to \mathbb{B}^{32}$	The special purpose register file.
$\mathsf{mm} :: \mathbb{B}^{32} \to \mathbb{B}^8$	The byte addressable memory.

The VAMP can be run in two different modes: system and user mode. The current mode is encoded in the register MODE of the spr. In system mode, programs can directly access the memory and fully control the architecture via a number of privileged instructions. In user mode, memory accesses are subject to address translation and attempts to execute a privileged instruction will result in an exception.

Transitions

The next state transition function of VAMP ISA is given by the function δ_{isa}, which takes a configuration and an external event vector and returns the successor configuration. The external event vector is a 19 bit-string, indicating for each connected device whether it is currently raising an interrupt signal or not:

$$\delta_{isa} :: C_{isa} \times \mathbb{B}^{19} \to C_{isa}$$

A transition can be sketched as follows: First, it is checked whether the current configuration (together with the external event vector) is causing an interrupt (see below). If not, the transition function is defined by a case-split on the instruction to which the delayed program counter is pointing (or more precisely: to the word in the memory pointed at by the dpc and which is interpreted as instruction).

Except for instructions related to interrupt handling and address translation the semantics of the VAMP ISA instruction set is equivalent to the one of VAMP assembly, and therefore omitted here.

Interrupt Handling

We distinguish two classes of interrupts: external and internal. External interrupts are caused by devices, and may be masked, i.e. disabled, by the processor by setting the special purpose register SR. This is used, for example, to ensure that an operating system kernel is never interrupted while running. Internal interrupts may be caused due to the execution of an illegal instruction or a privileged instructions in user mode, overflows during arithmetic operations, page-faults (see address translation) and by a special trap-instruction Itrap (used e.g. to invoke kernel calls).

The VAMP reacts to non-masked interrupts by entering into system mode and continuing execution at address 0, the start of the interrupt service routine (ISR). Normally, user mode is re-entered, continuing execution at the interrupted location, by issuing the privileged instruction Irfe, which marks the end of the ISR.

Address Translation

The VAMP provides a single-level address translation mechanism. The memory of the VAMP is partitioned into so called *pages*, small consecutive chunks of data. Each page has size *4K*, i.e. *4096* bytes. A memory address can be interpreted as follows: some bits denote the *page index* and the remaining bits the *byte index* in that page. In user mode, virtual memory addresses are translated via a page table to physical addresses. A page table is simply a mapping from virtual to physical page indices. It consists of word-sized page table entries, each entry contains (i) a physical page index, (ii) a write-protection bit, denoting whether the page is allowed to be written, and (iii) a valid bit, denoting whether the page is currently available in main memory or not. This bit may be used by a paging algorithm (cf. Chapter 4) to indicate that the requested page is currently swapped to the hard disk. The page table used by the hardware for translation is identified by the special purpose registers (i) page table origin, PTO, denoting its base address in memory, and (ii) the page table length, PTL.

A virtual address lookup may trigger a so called page-fault interrupt. This happens, for example, if the virtual page index lies outside the page table or the requested page is marked to be not valid.

3.1.3 Assembly

The VAMP assembly model provides a handy programming model by abstracting some features of the machine model VAMP ISA. In short, these abstractions are:

- Data is represented as integers, while addresses are represented as naturals instead of bit-vectors in VAMP ISA.

- VAMP assembly has a uniform, linear memory with no address translation visible. Hence, the model can be used to capture, either the execution of system mode computations or the execution of virtualized memory machines (i.e., for which memory virtualization is already implemented).

- Neither external nor internal interrupts are visible in VAMP assembly. This requires that all maskable interrupts are masked (i.e. the status register SR is set to zero) and that the execution of the VAMP assembly will not lead to any non-maskable interrupts.
- Instructions are no longer encoded as bit-vectors. Rather, an abstract data type encoding valid instructions is introduced. Correctness of instruction decoding has then to be shown once and for all.

VAMP Assembly Configuration

The VAMP assembly configuration is of record type C_{asm} with five fields:

pc :: $\mathbb{N}$	The program counter.
dpc :: $\mathbb{N}$	The delayed program counter, modeling the delayed branch mechanism.
gpr :: $\mathbb{Z}$ *list*	The general purpose register file, which is modeled as a list of integers.
spr :: $\mathbb{Z}$ *list*	The special purpose register file.
mm :: $\mathbb{N} \to \mathbb{Z}$	The main memory, which is a mapping from memory addresses (naturals) to memory cells (of type integer). This memory is to be interpreted as word-addressable memory of words.

Interpretation of Bit-Strings

In the underlying architecture data and addresses are represented as 32-bit strings. There are two ways to interpret these bit-strings, either as two's complement or as binary numbers.

In VAMP assembly we interpret data fields (memory and registers) as two's complement numbers and addresses as binary numbers. Thus, we type them as integers and naturals, respectively. We can switch between both interpretations by the two functions *i2n* and *n2i*:

$$i2n :: \mathbb{Z} \to \mathbb{N} \qquad\qquad n2i :: \mathbb{N} \to \mathbb{Z}$$

$$i2n(i) = \begin{cases} i + 2^{32} & \textbf{if } i < 0 \\ i & \textbf{otherwise} \end{cases} \qquad n2i(n) = \begin{cases} n - 2^{32} & \textbf{if } n > 2^{31} \\ n & \textbf{otherwise} \end{cases}$$

Since, the program is encoded in the main memory, instructions are also represented as integers. However, we translate these integers to the abstract data type $instr_T$. This translation is done by the function *to-instr*:

$$\textit{to-instr} :: \mathbb{Z} \to instr_T$$

Not all integers (or corresponding bit-strings on the ISA) decode to meaningful instructions. Thus, for some instructions, the translation may be undefined, i.e. return with *undef*.

Similarly, we can interpret a memory region starting at address *st* with length *l* as a list of instructions by the following function:

$$\mathit{to\text{-}instr\text{-}list} :: (\mathbb{N} \rightarrow \mathbb{Z}) \times \mathbb{N} \times \mathbb{N} \rightarrow \mathit{instr}_T\ \mathit{list}$$

$$\begin{array}{lll} \mathit{to\text{-}instr\text{-}list} & (m, st, 0) & = [] \\ & (m, st, l) & = (\mathit{to\text{-}instr}(m(st)))\#(\mathit{to\text{-}instr\text{-}list}(m, st+1, l-1)) \end{array}$$

By defining the transition system via a case distinction on the abstract data type instr_T, we no longer have to reason about instruction decoding, when proving correctness of assembly programs.

Validity of States

Not all VAMP assembly configurations encode meaningful states. For example, a register content with a number larger than $2^{31} - 1$ has no counterpart in hardware.

Therefore, we define the following two predicates to restrict the naturals and integers in the model to the domains of 32-bit binary numbers and 32-bit two's complement numbers.

The domain of 32-bit binary numbers and 32-bit two's complement numbers is defined by (cf. Notations):

$$\mathit{asm\text{-}nat} :: \mathbb{N} \rightarrow \mathit{bool} \qquad \mathit{asm\text{-}int} :: \mathbb{N} \rightarrow \mathit{bool}$$

$$\mathit{asm\text{-}nat}(n) = n \in \{0, \ldots, 2^{32} - 1\} \qquad \mathit{asm\text{-}int}(i) = i \in \{-2^{31}, \ldots, 2^{31} - 1\}$$

We call a VAMP assembly configuration *valid*, denoted by the predicate *valid-asm*, if the contents of the program counters, the register files and the memory obey the domain restrictions. Furthermore, the lists representing the register files must have 32 elements:

$$\mathit{valid\text{-}asm} :: C_{asm} \rightarrow \mathit{bool}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{valid\text{-}asm}(\mathit{asm}) \equiv} \\ & \mathit{asm\text{-}nat}(\mathit{asm}.\mathsf{pc}) \\ \wedge & \mathit{asm\text{-}nat}(\mathit{asm}.\mathsf{dpc}) \\ \wedge & (\forall \mathit{ind} < 32\,.\ \mathit{asm\text{-}int}(\mathit{asm}.\mathsf{gpr}!\mathit{ind})) \\ \wedge & (\forall \mathit{ind} < 32\,.\ \mathit{asm\text{-}int}(\mathit{asm}.\mathsf{spr}!\mathit{ind})) \\ \wedge & (\forall \mathit{ad}\,.\ \mathit{asm\text{-}int}(\mathit{asm}.\mathsf{mm}(\mathit{ad}))) \\ \wedge & \mathit{length}(\mathit{asm}.\mathsf{gpr}) = 32 \\ \wedge & \mathit{length}(\mathit{asm}.\mathsf{spr}) = 32 \end{array}$$

VAMP Assembly Transitions

The transition system of VAMP assembly is given by the next-step function:

$$\delta_{asm} :: C_{asm} \to C_{asm}$$

Given a configuration d it defines the successor state d' by a case distinction on the current instruction, that is the instruction to which the current delayed pc is pointing at:

$$current\text{-}instr :: C_{asm} \to instr_T$$

$$current\text{-}instr(asm) \equiv to\text{-}instr(asm.\mathsf{mm}(asm.\mathsf{dpc} \div 4))$$

We distinguish five kinds of instructions, whose semantics are described in the next paragraphs: (i) Arithmetical, logical and shift operations, (ii) memory operations, (iii) control operations, (iv) access to special purpose registers and, (v) interrupt related operations.

All instructions, except for control and interrupt, increment the delayed program counter by four (i.e. pointing to the next instruction) and set the new value of pc to the old one of the dpc:

$$inc\text{-}pc :: C_{asm} \to C_{asm}$$

$$\begin{aligned} inc\text{-}pc(asm) \equiv asm[&\mathsf{pc} := asm.\mathsf{dpc} +_{32} 4,\\ &\mathsf{dpc} := asm.\mathsf{pc}] \end{aligned}$$

In the following, we abbreviate states in which the program counters are incremented by $asm_{inc} = inc\text{-}pc(asm)$.

Arithmetic / Logical / Shift Operations The VAMP supports five types of arithmetic operations: addition and subtraction modulo 32, and the bitwise operations $\wedge_{32}$, $\vee_{32}$ and $\otimes_{32}$. For each operation two instructions are provided: either the second operand is a register content or it is the immediate constant of the instruction. Six shift operations are implemented: four logical and two arithmetic shifts. Moreover, the logical operators $=$,$\neq$,$<$,$>$,$\geq$ and $\leq$ are supported.

The table in Figure 3.1 gives the semantics of arithmetic and and shift instructions (note that we abbreviate $asm.\mathsf{gpr}!RS2$ by $RS2$). The table in Figure 3.2 gives the semantics of logical instructions.

Memory Operations The VAMP supports eight different memory access instructions: five for reading data from memory into the gpr, and three for storing data from the gpr to the memory. The effect of the instructions differ in size (words, half-words and bytes) and offset of the data to read/store.

Memory operations have three parameters:

Arithmetic Operations

Instruction	$\circ$	**OP2**
Iaddi RD RS i	$+^t_{32}$	i
Isubi RD RS i	$-^t_{32}$	i
Iandi RD RS i	$\wedge^t_{32}$	i
Iori RD RS i	$\vee^t_{32}$	i
Ixori RD RS i	$\otimes^t_{32}$	i
Iadd RD RS RS2	$+^t_{32}$	$RS2$
Isub RD RS RS2	$-^t_{32}$	$RS2$
Iand RD RS RS2	$\wedge^t_{32}$	$RS2$
Ior RD RS RS2	$\vee^t_{32}$	$RS2$
Ixor RD RS RS2	$\otimes^t_{32}$	$RS2$

Shift Operations

Instruction	$\circ$	**OP2**
Isll RD RS1 RS2	$<<^{t,l}_{32}$	$RS2$
Isrl RD RS1 RS2	$>>^{t,l}_{32}$	$RS2$
Isra RD RS1 RS2	$>>^{t,a}_{32}$	$RS2$
Islli RD rs sa	$<<^{t,l}_{32}$	sa
Isrli RD rs sa	$>>^{t,l}_{32}$	sa
Israi RD rs sa	$>>^{t,a}_{32}$	sa

Figure 3.1: Semantics of Arithmetic and Shift Operations: $asm' = asm_{inc}\,[\mathsf{gpr} := (asm.\mathsf{gpr}\,[RD := RS \circ OP2])]$

Logic Operations I
$OP2 = asm.\mathsf{gpr}!RS2$

Instruction	$\circ$
Iclr RD RS1 RS2	$\lambda ab.0$
Isgr RD RS1 RS2	$<$
Iseq RD RS1 RS2	$=$
Isge RD RS1 RS2	$\leq$
Isls RD RS1 RS2	$>$
Isne RD RS1 RS2	$\lambda ab.a \neq b$
Isle RD RS1 RS2	$\geq$
Iset RD RS1 RS2	$\lambda ab.1$

Logic Operations II
$OP2 = i$

Instruction	$\circ$
Iclri RD RS i	$\lambda ab.0$
Isgri RD RS i	$<$
Iseqi RD RS i	$=$
Isgei RD RS i	$\leq$
Islsi RD RS i	$>$
Isnei RD RS i	$\lambda ab.a \neq b$
Islei RD RS i	$\geq$
Iseti RD RS i	$\lambda ab.\mathsf{true}$

Figure 3.2: Semantics of Logical Operations: $asm' = asm_{inc}\,[\mathsf{gpr} := (asm.\mathsf{gpr}\,[RD := RS \circ OP2])]$

$RS1 \in \mathbb{N}$	The register from / to which the memory data is stored / loaded.
$RS2 \in \mathbb{N}$	The register containing the memory source or destination address in memory.
$i \in \mathbb{N}$	Offset of the memory address.

The set of all memory operations are identified by the following two predicates *is-store* and *is-load*:

$$\textit{is-store} :: C_{asm} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \textit{bool}$$

$$\begin{array}{l} \mathit{is\text{-}store}(\mathit{asm}, \mathit{RS1}, \mathit{RS2}, i) \equiv \\ \quad (\exists\, RS1, RS2, i\,. \\ \qquad \mathit{current\text{-}instr}(\mathit{asm}) \in \{ \ \text{Ilw RS1 RS2 i}, \text{Ilb RS1 RS2 i}, \text{Ilh RS1 RS2 i}, \\ \qquad\qquad \text{Ilbu RS1 RS2 i}, \text{Ilhu RS1 RS2 i}\}) \end{array}$$

$$\mathit{is\text{-}load} :: C_{asm} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathit{bool}$$

$$\begin{array}{l} \mathit{is\text{-}load}(\mathit{asm}, RS1, RS2, i) \equiv \\ \quad (\exists\, RS1, RS2, i\,. \\ \quad \mathit{current\text{-}instr}(\mathit{asm}) \in \{\text{Isb RS1 RS2 i}, \text{Ish RS1 RS2 i}, \text{Isw RS1 RS2 i}\}) \end{array}$$

The effective address of store / load operations is computed as follows:

$$\mathit{ea} :: C_{asm} \rightarrow \mathbb{N}$$

$$\mathit{ea}(\mathit{asm}) \equiv \begin{cases} \mathit{i2n}(\mathit{asm}.\mathsf{gpr}!RS2) +_{32} \mathit{i2n}(i) & \textbf{if} \ \ \mathit{is\text{-}store}(\mathit{asm}, RS2, ?, i) \vee \\ & \quad\ \ \mathit{is\text{-}load}(\mathit{asm}, RS2, ?, i) \\ \mathit{undef} & \textbf{otherwise} \end{cases}$$

In the remainder of this article, only instructions for reading and writing complete words will be used. The semantics of these instructions is only defined if the memory address is well formed, i.e. it has to be in range and word aligned:

$$\mathit{valid\text{-}ad} :: \mathbb{N} \rightarrow \mathit{bool}$$

$$\mathit{valid\text{-}ad}(\mathit{ad}) = \mathit{ad} + 4 < 2^{32} \wedge 4 \ \mathsf{dvd} \ \mathit{ad}$$

We get for reading and writing words the following semantics:

$$\frac{\mathit{current\text{-}instr}(\mathit{asm}) = \text{Ilw RS1 RS2 i} \qquad \mathit{valid\text{-}ad}(\mathit{ea}(\mathit{asm}))}{\mathit{asm}' = \mathit{asm}_{inc}[\mathsf{gpr} := (\mathit{asm}.\mathsf{gpr}[RS1 := \mathit{asm}.\mathsf{mm}(\mathit{ea}(\mathit{asm}) \div 4)])]}$$

$$\frac{\mathit{current\text{-}instr}(\mathit{asm}) = \text{Isw RS1 RS2 i} \qquad \mathit{valid\text{-}ad}(\mathit{ea}(\mathit{asm}))}{\mathit{asm}' = \mathit{asm}_{inc}[\mathsf{mm} := \mathit{asm}.\mathsf{mm}((\mathit{ea}(\mathit{asm}) \div 4) := \mathit{asm}.\mathsf{gpr}!RS1)]}$$

In the following we use the predicates $\mathit{lw}(\mathit{asm}) \equiv \mathit{current\text{-}instr}(\mathit{asm}) =$ Ilw ? ? ? and $\mathit{sw}(\mathit{asm}) \equiv \mathit{current\text{-}instr}(\mathit{asm}) =$ Isw ? ? ? to indicate a load or store word instruction.

Control The VAMP supports six control instructions: two branch operations and four unconditional jumps. All of them have the same effect on the delayed program counter: $\mathit{asm}'.\mathsf{dpc} = \mathit{asm}.\mathsf{pc}$. Jumps are either relative or absolute, and may either save the incremented current pc to a predefined register L or not. The effects of the control instructions on the pc and possibly on

Control Operations

Instruction	**New Program Counter** pc_n	**Save Register** $data_n$
Ibeqz R i	$\begin{cases} asm.\mathsf{pc} +_{32} i & asm.R = 0 \\ asm.\mathsf{pc} +_{32} 4 & \textbf{otherwise} \end{cases}$	$asm.\mathsf{gpr!L}$
Ibnez R i	$\begin{cases} asm.\mathsf{pc} +_{32} i & asm.R \neq 0 \\ asm.\mathsf{pc} +_{32} 4 & \textbf{otherwise} \end{cases}$	"
Ij i	$asm.\mathsf{pc} +_{32} i$	"
Ijr R	$i2n(asm.R)$	"
Ijalr R	$i2n(asm.R)$	$n2i(asm.\mathsf{pc} +_{32} 4)$
Ijal i	$asm.\mathsf{pc} +_{32} i$	"

Table 3.1: Control instructions

the gpr are summarized by the following formula, where the variables pc_n and $data_n$ are defined in Table 3.1 (note that we abbreviate $asm.\mathsf{gpr}!R$ by $asm.R$):

$$\begin{aligned} asm' = asm[&\mathsf{dpc} := asm.\mathsf{pc}, \\ &\mathsf{pc} := pc_n, \\ &\mathsf{gpr} := asm.\mathsf{gpr}[\mathsf{L} := data_n]] \end{aligned}$$

Move to/from spr The two instructions Imovs2i RD SA and Imovi2s SA RS copy data between a general purpose register and a special purpose register. Both instructions can only be executed in system mode. Otherwise no effect is visible in the assembly model:

$$\frac{\textit{current-instr}(asm) = \text{Imovs2i RD SA} \qquad asm.\mathsf{spr!MODE} = 1}{asm' = asm_{inc}[\mathsf{gpr} := asm.\mathsf{gpr}[RD := asm.\mathsf{spr}!(SA)]]}$$

$$\frac{\textit{current-instr}(asm) = \text{Imovs2i RD SA} \qquad asm.\mathsf{spr!MODE} = 1}{asm' = asm_{inc}[\mathsf{spr} := asm.\mathsf{spr}[SA := asm.\mathsf{gpr}!(RD)]]}$$

Interrupts The VAMP supports two instructions dealing with interrupt handling. The first is Itrap, which triggers a trap interrupt, and the second is Irfe, which is invoked when returning from an interrupt handler. Since neither interrupt triggering, nor interrupt handling is modeled in VAMP assembly, also the corresponding instructions should not have any visible effects. Still, they have to be modeled and they may be part of a valid VAMP assembly program.

$$\frac{\textit{current-instr}(asm) = \text{Irfe}}{asm' = asm} \qquad \frac{\textit{current-instr}(asm) = \text{Itrap ?}}{asm' = asm}$$

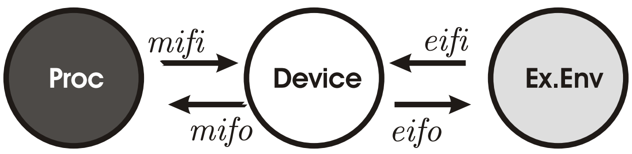


Figure 3.3: Interaction of device with processor and external Environment

VAMP Assembly Computations

The function Δ_{asm} denotes a t-step VAMP assembly computation. We have $\Delta_{asm}(asm, t+1) = \Delta_{asm}(\delta_{asm}(asm), t)$ if $t > 0$ and asm otherwise.

3.1.4 Integrating Devices

In this section we first introduce our general device model, then we sketch the device integration into the VAMP gate-level and VAMP ISA model, and finally present the desired programming model VAMP assembly with devices in detail.

Modeling a Device

Devices are modeled as finite state transition systems interacting with the processor and with an external environment. The latter may be used, for example, to model user interaction or a network.

Communication with the processor is modeled via so called memory interface input, *mifi*, and output, *mifo*. The naming is relative to the viewpoint of the device, i.e. a *mifi* encodes a request of the processor to the device, and a *mifo* the corresponding answer of the device to the processor (see Figure 3.3).

A *mifi* is of record type Mifi_T, which has the following four fields:

$\mathsf{rd} \in \mathit{bool}$	The read flag signals a read request of the processor.
$\mathsf{wr} \in \mathit{bool}$	The write flag signals a read request of the processor.
$\mathsf{ad} \in \mathbb{N}$	The address encodes the device port from / to which the processor wants to read / write.
$\mathsf{din} \in \mathbb{N}$	In case of a write request, this field encodes the data (one word) to be written from the processor to the device. Note, that (in contrast to assembly) data is interpreted as a binary number.

We call the element $(\mathsf{rd} = \mathsf{false}, \mathsf{wr} = \mathsf{false}, \mathsf{ad} = 0, \mathsf{din} = 0)$ the *idle mifi* and denote it with mifi_ϵ. A memory output $\mathit{mifo} \in \mathit{Mifo}_T$ of a device contains the result of a requested read input from the processor. This output encodes a word as natural, i.e. $\mathit{Mifo}_T = \mathbb{N}$. The interface to the external environment depends on the type of each device.

In the following let X denote a name of a specific kind / type of device (e.g. X is later on instantiated by a hard disk or a serial interface). Given the device type X, a device transition system is defined by the following three components:

S_X	The state space of the device.
$\delta_X :: \; S_X \times \mathit{Eifi}_X \times \mathit{Mifi}_T \rightarrow S_X \times \mathit{Mifo}_T \times \mathit{Eifo}_X$	The transition function takes a device state, an input from the external environment of type Eifi_X, and an input from the processor. It returns the next state, an output to the processor and an output to the external environment of type Eifo_X.
$ir_X :: S_X \rightarrow \mathit{bool}$	The interrupt function denotes, whether the device is currently signaling an interrupt.

In the models considered here a device either consumes an external or a processor input, never both simultaneously. Hence, in a step either *eifi* or *mifi* is empty, denoted with eifi_ϵ and mifi_ϵ.

Assuming that the state space of different device types is disjoint, we can define a mapping from device states to the corresponding type.[1] For such a mapping *dsty* we have:

$$\mathit{dsty}(s) = X \Leftrightarrow s \in S_X$$

The set of identifiers of devices connected to the processor is denoted by $\mathfrak{D} = [0, \ldots, 7]$. That means, we support up to eight devices. We relate device identifiers to device type names by the function *dty*. Let, moreover, the set $S_{\mathfrak{D}}$ denote the union on all possible device states, and the sets $\mathit{Eifi}_{\mathfrak{D}}$ and $\mathit{Eifo}_{\mathfrak{D}}$ denote the union of all possible external input and output types.

So far three devices have been formalized in Isabelle/HOL: the hard disk [AH08], the serial interface [AHK$^+$07], and an automotive bus controller [Kna08, Böh07]. The latter has been even formalized at the level of gates, such that a precise scheduling correctness theorem could be proven [ABK08].

VAMP Gate-level with devices

Devices are mapped into the memory address space, i.e. they can be accessed by regular load and store word operations (see Figure 3.4). Addresses reserved to devices can be identified by leading ones at the positions 31 to 15. Furthermore, each device has its own address domain. Address domains of different devices are mutually exclusive and are identified by the bits 14, 13

[1] In Isabelle device types are implemented via an abstract data type, with one constructor for each type.

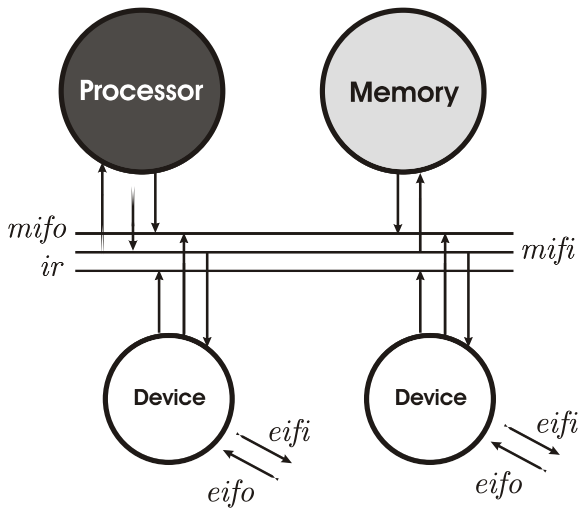


Figure 3.4: The VAMP, the memory and devices

and 12 of the memory address (for formal definitions, see VAMP assembly with devices).

At the level of gates the processor and all connected devices are executed in parallel. Hence, in each cycle, all devices may take a step and consume input from the external environment. Then, a computation can be defined by the initial global configuration and an input sequence of the external environment, which maps cycle numbers to device inputs:

$$input_T = \mathbb{N} \rightarrow \mathfrak{D} \rightarrow Eifi_{\mathfrak{D}}$$

A formal definition of the transition system is omitted and can be found in [Tve09]. Here, we only sketch two design decisions of the VAMP, which are crucial for device integration:

- If an instruction is interrupted, the effects of all instructions which are in the pipeline and are older than the interrupted instruction have to be rolled back. This, however, is not always possible: Interrupts are sampled in the write-back stage, whereas a device access becomes already visible (i.e. modifies the device) in the memory stage and can not be roll backed. The solution is to stall, i.e. defer the execution of any device access (as any other memory access) until it becomes the oldest instruction in the pipe.
- An instruction may interfere with a device twice: once in the memory stage, in case of a load or store word instruction to or from the device's

address domain, and once in the write-back stage, if the device triggers an interrupt which is not masked. Thus, during one instruction travels the pipe, it may 'see' two different device states. Such a construction could lead to undesired artifacts: An instruction may deactivate an interrupt signal of a device in the memory stage, where a new interrupt is triggered by the device before the instruction reaches the write-back stage. The solution is to sample device interrupts already in the memory stage and to make them visible for the processor in the write-back stage.

VAMP ISA with devices

The instruction set architecture abstracts from cycles to execution of complete instructions. Unfortunately, there is no fixed mapping from instructions to the number of cycles needed to execute this instruction on the hardware. For example, the duration of memory accesses of the same instruction may vary in order of magnitudes due to different cache content, which is no longer visible in ISA. Hence, moving to ISA means also a loss of timing information. This lack of granularity makes it impossible to determine how many steps the device has taken during the execution of one instruction. We compensate for this loss by introducing a concurrent model, in which devices and the processor are executed in an interleaved way.

Configurations of the model VAMP ISA with devices are of record type C_{isad} with the following two fields:

$\mathsf{proc} \in C_{isa}$	The VAMP ISA configuration of the processor.
$\mathsf{devs} \in \mathfrak{D} \to S_{\mathfrak{D}}$	The configuration of all connected devices.

In the following we refer to such configurations as global ISA configurations.

The transition function of VAMP ISA with devices, denoted by δ_{isad}, has to distinguish whether the processor or a device executes next. Therefore, it takes the current global configuration and an oracle, called *event*. The event equals P in case of a processor step or is a pair (id_D, eifi) of device identifier and environment input in case of a device step. Thus, an event is of type $event_T = \{P\} \cup (\mathfrak{D} \times \mathit{Eifi}_{\mathfrak{D}})$. The transition function returns the next global state and an external output:

$$\delta_{isad} :: C_{isad} \times event_T \to C_{isad} \times \mathit{Eifo}_{\mathfrak{D}}$$

Devices and the processor are executed in an interleaved way. A model run is defined by the start configuration and an *execution sequence*. The latter returns for a given step number the oracle event input, i.e., it resolves the non-determinism. It is of type:

$$Seq_T = \mathbb{N} \to event_T$$

Note, that we only consider execution sequences which are well typed, i.e. where the device type and the input from the environment match:

$$\mathit{well\text{-}typed}(seq) \equiv (\forall i. seq(i) \neq P \implies snd(seq(i)) \in \mathit{Eifi}_{dty(fst(seq(i)))})$$

The function Δ_{isad} is used to model a computation of the overall system. It takes a global start configuration, an execution sequence and a step number n as inputs. It returns a pair consisting of the global configuration reached after applying the transition function δ_{isad} for n times and the sequence of external output generated during this process:

$$\Delta_{isad} :: C_{isad} \times Seq_T \times \mathbb{N} \to C_{isad} \times (\mathit{Eifo}_{\mathfrak{D}}\ \mathit{list})$$

A detailed description of the transition functions δ_{isad} and Δ_{isad} is omitted and can be found in [AHK+07]. Instead, we elaborate in the next paragraph on the counterparts of these functions in the model VAMP assembly with devices.

VAMP assembly with devices

Configurations of the model VAMP assembly with devices are of record type C_{asmd}, with the following two fields:

$\mathsf{proc} \in C_{asm}$	The VAMP assembly configuration of the processor.
$\mathsf{devs} \in \mathfrak{D} \to S_{\mathfrak{D}}$	The configuration of all connected devices.

In the following we refer to such configurations as global assembly configurations.

Each device with identifier $id_D \in \mathfrak{D}$ is mapped into the processor's memory at address ranges $DA_{id_D} \subset \mathbb{N}$. Different device address ranges are mutually exclusive. We denote with DA the union of all device addresses, i.e. $DA = \bigcup_{id_D \in \mathfrak{D}} DA_{id_D}$.

On the bit-level representation, addresses reserved to devices can be identified by leading ones at the positions 31 to 15. Thus, we have:

$$\mathsf{device\text{-}border} = \sum_{i=15}^{i \leq 31} 2^i = 4294934528$$

$$a \in DA \Leftrightarrow a \geq \mathsf{device\text{-}border}$$

Furthermore, on bit-level, the address domains DA_{id_D} of devices are identified by the bits 14, 13 and 12 of the memory address. The following function returns for a global assembly configuration either P in case the current instruction is not a device access, or the index of the device to which the accessed address belongs:

$$da :: C_{asm} \to \{P\} \cup \mathfrak{D}$$

$$da(asm) = \begin{cases} (ea(asm) \mod 2^{15}) \div 2^{12} & \textbf{if} \quad ea(asm) \neq undef \\ & \quad \wedge \; ea(asm) \in DA \\ P & \textbf{otherwise} \end{cases}$$

Similarly, the accessed port is encoded in the bits 2 to 11 of the memory address. We define the function *port* as follows:

$$port :: \mathbb{N} \to \mathbb{N}$$

$$port(a) = (a \mod 2^{12}) \div 4$$

Moreover, we denote the address of the first port of a device id_D by the following function:

$$D0 :: \mathfrak{D} \to \mathbb{N}$$

$$D0(id_D) \equiv \mathsf{device\text{-}border} + 2^{12} \cdot id_D$$

Next, we define the output *mifi* generated by the processor in some configuration *asm* as follows:

$$mifi :: C_{asm} \to Mifi_T$$

```
$mifi(asm) =$
  if $da(asm) = P$ then $mifi_\epsilon$ else
  (case $current\text{-}instr(asm)$ of
      (Ilw RS1 RS2 i)   $\Longrightarrow (\mathsf{true}, \mathsf{false}, port(ea(asm)), 0)$
      (Isw RS1 RS2 i)   $\Longrightarrow (\mathsf{false}, \mathsf{true}, port(ea(asm)), i2n(asm.\mathsf{gpr}!RS1))$
      _                 $\Longrightarrow mifi_\epsilon$)
```

Like in the ISA case, the transition system of VAMP assembly with devices, takes as input the current global configuration $asmd = (\mathsf{proc} = asm, \mathsf{devs} = ds)$ and the event ev, which distinguished between a processor / device step and provides external input for the latter case. It returns the next global configuration $asmd'$ and the output $eifo$:

$$\delta_{\text{asmd}} :: C_{\text{asmd}} \times event_T \to C_{\text{asmd}} \times Eifo_{\mathfrak{D}}$$

The transition function is defined via the following case distinction:

- A processor-device transition is taken if it is the processor's turn and the current instruction accesses a device with index id_D. We express this case formally by:

  $$ev = P \wedge da(asm) = id_D$$

  The device takes a step with the transition function corresponding to its type, and consumes the *mifi* request generated by the processor and an empty external input. In case of a read request (i.e. of a load word operation), the device returns a *mifo* to the processor, and the processor configuration is updated by writing this output back to the effective address of the current instruction and by incrementing the program counters. In case of a write request (i.e. a store word operation), only the program counters are incremented. Formally, the output $(asmd', eifo)$ of the transition is given by:

```
let
    $X = dty(id_D)$
    $mifi = mifi(asm)$
    $(dx', mifo, eifo) = \delta_X(ds(id_D), eifi_\epsilon, mifi)$
    $asm' = inc\text{-}pc(asm)$
    $asm'' = \begin{cases} asm'[\mathsf{mm} := \mathsf{mm}((ea(asm) \div 4) := mifo)] & \textbf{if } mifi.\mathsf{wr} \\ asm' & \textbf{otherwise} \end{cases}$
in
    $((\mathsf{proc} = asm'', \mathsf{devs} = ds(id_D := dx')), eifo)$
```

- A local processor transition is taken if it is the processor's turn, and the current instruction does not access a device. We express this case formally by:

$$ev = P \wedge da(asm) = P$$

The processor configuration is updated by applying the VAMP assembly transition function to the current processor configuration. The output to the external environment is empty. Formally, the output $(asmd', eifo)$ of the transition is given by:

$$((\mathsf{proc} = \delta_{asm}(asm), \mathsf{devs} = ds), eifo_\epsilon)$$

- An external device transition is taken if the event schedules some device execution:

$$ev = (id_D, eifi)$$

Only the configuration of device id_D is updated by applying the transition function corresponding to type of device id_D. The input from the processor is set to be the empty input. Formally, the output $(asmd', eifo)$ of the transition is given by:

```
let
    $X = dty(id_D)$
    $(dx', mifo, eifo) = \delta_X(ds(id_D), eifi, mifi_\epsilon)$
in
    $((\mathsf{proc} = asm, \mathsf{devs} = ds(id_D := dx')), eifo)$
```

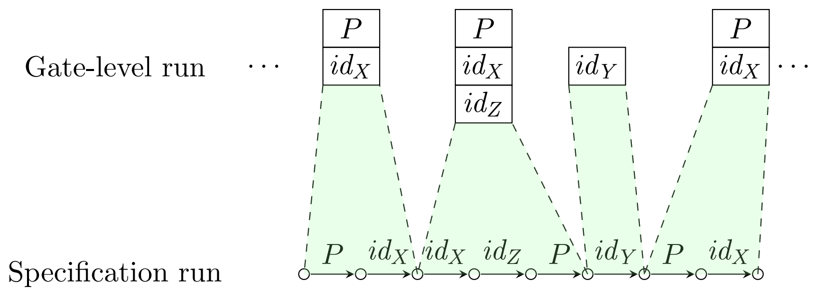


Figure 3.5: Example for the scheduling function sI_{PD}

Note, that regular memory access is now restricted to addresses which are smaller than the address domain, i.e. we redefine the predicate *valid-ad* to:

$$\mathit{valid\text{-}ad}(ad) = ad + 4 < \mathsf{device\text{-}border} \wedge 4 \;\mathsf{dvd}\; ad$$

As in VAMP ISA with devices, the processor and devices are executed in an interleaved way. The function Δ_{asmd} is used to model a computation or run of the overall system. It takes a global start configuration, an execution sequence and a step number n as inputs. It returns a pair, the global configuration reached after applying the transition function for n times and the sequence of external output generated during this process:

$$\Delta_{\text{asmd}} :: C_{\text{asmd}} \times \mathit{Seq}_T \times \mathbb{N} \to C_{\text{asmd}} \times (\mathit{Eifo}_{\mathfrak{D}}\ \mathit{list})$$

$$\begin{array}{lll}
\Delta_{\text{asmd}} & (\mathit{asmd}, \mathit{seq}, 0) & = (\mathit{asmd}, [\,]) \\
& (\mathit{asmd}, \mathit{seq}, n+1) & = \\
& & \quad \textbf{let} \\
& & \quad\quad (\mathit{asmd}', \mathit{eifo}_l) = \Delta_{\text{asmd}}(\mathit{asmd}, \mathit{seq}, n) \\
& & \quad\quad (\mathit{asmd}'', \mathit{eifo}) = \delta_{\text{asmd}}(\mathit{asmd}', \mathit{seq}(n+1)) \\
& & \quad \textbf{in} \\
& & \quad\quad (\mathit{asmd}'', \mathit{eifo} \# \mathit{eifo}_l)
\end{array}$$

3.1.5 Simulation Theorems

From ISA to gate-level

By using a simulation theorem, properties proven in VAMP ISA with devices can be transfered to VAMP gate-level with devices. Note, that the first model is non-deterministic, whereas the hardware is not (once the input sequence is known).

The abstraction relation consists of two parts: processor and device abstraction. The first relation maps all programmer visible registers and the memory to their counterparts in the instruction set architecture. The devices abstraction is provided as parameter to the theorem (in the simplest case it is the identity).

The concrete formulation of the theorem and the corresponding proof can be found in [Tve09]. In short, the theorem claims that for each computation on gate-level, a VAMP ISA with devices computation exists, which simulates the first. To resolve the non-determinism in VAMP ISA with devices, we have to construct for each hardware run a corresponding execution sequence. This construction is provided by the *device scheduling function*, which is denoted by sI_{PD} (cf. Figure 3.1.5). It takes a configuration of the hardware, an input sequence, and returns an execution sequence:[2]

$$sI_{PD} :: C_h \times input_T \to Seq_T$$

The scheduling function is of great importance, since it encodes all timing information lost when moving from gate-level to ISA. By that, it also restricts the domain of execution sequences which we have to consider when proving correctness of some property over the concurrent model: For translating a property from VAMP ISA with devices down to the hardware, it suffices to verify the property only for those execution sequences that may be generated by the scheduling function. Execution sequences which lay outside the domain of the scheduling function do also not correspond to any hardware run, and hence are irrelevant for the correctness of any hardware property.

Surely, reasoning about the scheduling function each time when verifying a program at ISA would make the whole processor abstraction obsolete. Instead, we identify a set of general properties over the range of the scheduling function, which also serve as restrictions for execution sequences. For example, termination of drivers can typically only be shown if processor and devices are scheduled infinitely often:

$$\begin{aligned} \textit{proc-live}(seq) &\equiv \textit{live}(seq, \lambda ev.\, ev = P) \\ \textit{dev-live}(seq) &\equiv \forall id_D \in \mathfrak{D}.\, \textit{live}(seq, \lambda ev.\, ev = (id_D, ?)) \end{aligned}$$

The *set of valid sequences* contains all well-typed sequences (cf. Section 3.1.4) which comply to both liveness conditions defined above:

$$Seq_V = \{seq \in Seq_T \mid \textit{well-typed}(seq) \wedge \textit{proc-live}(seq) \wedge \textit{dev-live}(seq)\}$$

Lemma 2 (Liveness of Sequences)

$$\textit{range}(sI_{PD}) \subseteq Seq_V$$

The proof is conducted at gate-level hardware. So far, the formal liveness proof for the VAMP has still to be mechanized in Isabelle/HOL. Liveness of device steps is assumed as precondition in the hardware correctness theorem [Tsy09] and has to be discharged for the concrete device implementation.

[2]The formal definition in [Tve09] differs a bit in the signature, as it takes an additional cycle number and returns also a corresponding step number.

Correctness of drivers could depend on further device-specific restrictions of the environment. For example, for the hard disk the environment eventually signals termination of a read or write operation. Such assumptions are also formulated in terms of Seq_V and proven for the scheduling function.

From Assembly to ISA

By using a simulation theorem, results proven in VAMP assembly with devices can be transfered to VAMP ISA with devices and subsequently to the gate-level implementation of the hardware. VAMP assembly with devices may be used to model the execution of the hardware either in system mode, or in user mode in case virtualization has been implemented. Each application requires a different simulation theorem. In the remainder we will only use and therefore state the theorem in case of system mode.

To establish simulation we first define an abstraction relation between the global configurations of both models, and then prove a step-by-step simulation theorem. Note, however, that not all ISA computations can be expressed in assembly, as some abstractions reduce the expressiveness of the computational model. Moreover, not all assembly configurations encode valid ISA states. Thus, we need also to define the set of preconditions under which simulation is valid. A detailed proof of the transfer theorem is reported in [Tsy09].

Abstraction relation. Global ISA configurations are related to global assembly configurations by the following predicate:

$$\sim_{\textit{isa-asm}} :: C_{isad} \times C_{\mathrm{asmd}} \rightarrow \textit{bool}$$

In short, it is defined as follows:

- Both processor configurations (ISA and assembly) consist of the same components, but as their types are different we relate them by a type conversion. The abstraction relation, requires that (i) the program counters in ISA interpreted as binary numbers must be equal to the assembly program counters, and, (ii) the memory and register contents in ISA interpreted as two's complement numbers are equal to the content of memory and registers in assembly.[3]
- Devices are related by equality, since in both models the same device abstraction is used.

Preconditions for simulation. Not all ISA computations can be simulated in assembly, and not every assembly state does encode a valid ISA state.

[3]Some of the special purpose registers are ignored by the abstraction relation, since they have no visible effect in VAMP assembly.

These two type of restrictions, dynamic properties on a computation and static conditions on the initial states are both expressed on the level of assembly.

For the initial state we require that its integers and naturals representing data and addresses are well-typed, i.e. in range, and that the code region does not overlap with the device domain. While the code is not explicitly marked in the assembly configuration, we can identify the corresponding region by the two parameters *start-ad*, for the start address (in words) of the code in the memory and *prog-len*, for the length (in words) of the program. We give meaning to the parameters, by assuming that during a computation, only instructions from the corresponding memory region are read. In the following we abbreviate a code-range by the following function:

$$\textit{code-range}(\textit{start-ad}, \textit{prog-len}) = [4 \cdot \textit{start-ad}, \ldots, 4 \cdot (\textit{start-ad} + \textit{prog-len})]$$

We define the initial condition for simulation by:

$$\textit{isa-asm-precond}_{\textit{init}} :: C_{\textit{asm}} \times \mathbb{N} \times \mathbb{N} \to \textit{bool}$$

$$\begin{aligned} &\textit{isa-asm-precond}_{\textit{init}}(\textit{asm}, \textit{start-ad}, \textit{prog-len}) = \\ &\quad \textit{valid-asm}(\textit{asm}) \\ \wedge &\quad 4 \cdot (\textit{start-ad} + \textit{prog-len} - 1) < \textsf{device-border} \end{aligned}$$

For the dynamic conditions, we require that during an assembly computation i) the mode is always set to system mode, ii) the status register remains zero, i.e. all interrupts stay masked, iii) neither of the instructions Irfe and Itrap is executed, since neither mode switches nor the interrupt mechanism is modeled in assembly, iv) instructions are only fetched from the predefined program range, v) the code range in memory is never written, i.e. no self-modifying code is permitted, vi) the addresses from which instructions are fetched are always word aligned, and, vii) word access of devices is ensured.

$$\textit{step-properties} :: C_{\textit{asm}} \times \mathbb{N} \times \mathbb{N} \to \textit{bool}$$

$$\begin{aligned} &\textit{step-properties}(\textit{asm}, \textit{start-ad}, \textit{prog-len}) == \\ &\quad \textit{asm}.\textsf{spr!MODE} = 0 \\ \wedge &\quad \textit{asm}.\textsf{spr!SR} = 0 \\ \wedge &\quad \textit{current-instr}(\textit{asm}) \notin \{\text{Itrap }?, \text{Irfe}\} \\ \wedge &\quad \textit{asm}.\textsf{dpc} \in \textit{code-range}(\textit{start-ad}, \textit{prog-len} - 1) \\ \wedge &\quad \textit{is-store}(\textit{asm}, ?, ?, ?) \implies \textit{ea}(\textit{asm}) \notin \textit{code-range}(\textit{start-ad}, \textit{prog-len}) \\ \wedge &\quad 4 \ \textsf{dvd} \ \textit{asm}.\textsf{dpc} \\ \wedge &\quad (\textit{is-store}(\textit{asm}, ?, ?, ?) \wedge \textit{ea}(\textit{asm}) \in \textit{DA}) \implies \textit{lw}(\textit{asm}) \vee \textit{sw}(\textit{asm}) \end{aligned}$$

We require for VAMP assembly computations that *step-properties* holds in each step:

$$\textit{isa-asm-precond}^{p}_{\textit{dyn}} :: C_{\textit{asm}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \textit{bool}$$

$$\begin{gathered}\textit{isa-asm-precond}^{p}_{dyn}(asm_0, \textit{start-ad}, \textit{prog-len}, N) = \\ (\forall n \leq N \,.\, \textit{step-properties}(\Delta_{asm}(asm_0, n), \textit{start-ad}, \textit{prog-len}))\end{gathered}$$

Similarly, we can extend the definition of dynamic preconditions on computations for VAMP assembly with devices:

$$\textit{isa-asm-precond}_{dyn} :: C_{\text{asmd}} \times \mathbb{N} \times \mathbb{N} \times \textit{Seq}_T \times \mathbb{N} \rightarrow \textit{bool}$$

$$\begin{gathered}\textit{isa-asm-precond}_{dyn}(asmd_0, \textit{start-ad}, \textit{prog-len}, seq, N) = \\ \forall n \leq N \,.\, \textit{step-properties}(\textit{fst}(\Delta(asmd_0, seq, n)).\mathsf{proc}, \textit{start-ad}, \textit{prog-len})\end{gathered}$$

Stating the Theorem. An ordinary step-by-step simulation holds between VAMP ISA and VAMP assembly, where it suffices to prove the theorem only for valid sequences as defined in the simulation theorem between VAMP ISA with devices and VAMP gate-level with devices:

Theorem 1 (VAMP assembly to VAMP ISA)

$$\begin{array}{ll} \multicolumn{2}{l}{\forall seq \in \textit{Seq}_V \,.} \\ & \textit{isa-asm-precond}_{init}(asmd.\mathsf{proc}, \textit{start-ad}, \textit{prog-len}) \\ \wedge & \textit{isa-asm-precond}_{dyn}(asmd, \textit{start-ad}, \textit{prog-len}, seq, N) \\ \wedge & isad \sim_{\textit{isa-asm}} asmd \\ \Longrightarrow & \mathit{\Delta}_{isad}(isad, seq, N) \sim_{\textit{isa-asm}} \mathit{\Delta}_{asmd}(asmd, seq, N) \end{array}$$

This theorem has benn formally proven by Tsyban [Tsy09]. Thus, again, for proving properties on VAMP assembly with devices only valid executions sequences have to be considered.

3.2 VAMP Reordered

Obviously, when proving correctness of a concrete driver for a specific device, an interleaved semantics of all devices is cumbersome. Preferably, for the proof we would like to use a simpler programming model first, e.g., a sequential model or a model with just a single device, and then generalize the result. In this section we develop theory for that purpose, on top of the model VAMP assembly with devices. In the following we refer to this model by calling it *the combined model*.

We assume that we have driver code that exclusively controls a certain device $id_D \in \mathfrak{D}$ and only that device. By choosing the combined model as the programming model, we implicitly assume that all interrupts are masked in hardware via the special-purpose status register while the driver is running. In our scenario this restriction is not severe. On the one hand, interrupts of device id_D are assumed to be already delivered to our driver. On the other hand, interrupts for other devices should be handled by different drivers in a manner transparent to the driver under verification. Thus, this problem is orthogonal to the one that we focus on here. Techniques for the verification of concurrent (assembly) programs apply in this case (cf. [YS04]).

A key observation in our scenario is that some steps in the computation of the combined model can be swapped without changing the outcome. This reordering is sound if devices do not influence each other. By swapping steps repeatedly, an execution of the driver in the combined model can be separated into steps involving only the driver and the controlled device followed by steps involving only other devices. Thus, correctness of the driver can be shown in a model with only the processor and the controlled device. Still, this model is concurrent. Two further simplifications may be applicable for parts of the driver execution. First, for phases not involving device access at all properties can be proven relative to the isolated processor model, only. Second, for phases in which the device is in a stable state (by which we mean it does not react to external input) properties can be proven relative to a model without (external) device steps.

A similar technique can be applied for higher-level models with devices. For example, in Verisoft the bulk of all software is implemented in C0 (cf. Section 3.3). Most of this code is verified in a Hoare logic verification environment for C0. Integration of concurrent correctness results into traditional Hoare logic proofs is hardly manageable (concurrent logics have been used for example in Microsoft's Verifying C Compiler [CMST09]). It is much more convenient to show the correctness of high-level code against a sequential specification in which the driver calls are executed atomically. Because we use type-safe C0 we cannot allow direct access to device ports, which do not behave like regular memory. Hence, reordering techniques can be used for a concurrent C0 semantics with devices, separating C0 steps from device and driver execution steps. The theory developed here is applied to C0 verification

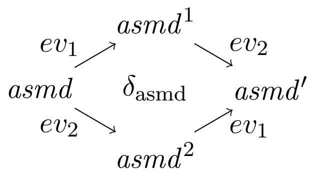


Figure 3.6: Swapping Two Non-Interfering Steps

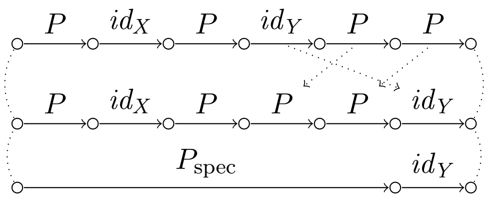


Figure 3.7: Reordering and Abstraction of an Execution Sequence

in the subsequent section.

3.2.1 A Basic Observation

A basic observation of our overall model is that device and processor steps not interfering with each other can be swapped. We say that two steps do not interfere if at least one is not a processor step and if they do not involve the same device; we call a device involved in a step if it is accessed by the processor or makes a step itself.

Recall that for a processor configuration *asm* the function *da* indicates whether the processor makes a local step, $da(asm) = P$, or accesses a specific device, $da(asm) \in \mathfrak{D}$. For an event *ev*, we let the function *Da* denote the *set* of components involved in a step:

$$Da :: asm \times event_T \rightarrow pow(\{P\} \cup \mathfrak{D})$$

We have $P \in Da(asm, ev)$ iff $ev = P$ and $id_D \in Da(asm, ev)$ iff $ev = (id_D, ?)$ or $da(asm) = id_D$.

Lemma 3 (Non-Interference Observation) *For a global VAMP assembly configuration asmd, two events ev_1 and ev_2 with $Da(asmd.\mathsf{proc}, ev_1) \cap Da(asmd.\mathsf{proc}, ev_2) = \emptyset$ can be executed in arbitrary order, as depicted in Fig. 3.6:*

$$fst(\delta_{asmd}(fst(\delta_{asmd}(asmd, ev_1), ev_2))) = fst(\delta_{asmd}(fst(\delta_{asmd}(asmd, ev_2), ev_1)))$$

The lemma is proven by a simple expanding of definitions.

Validity of Observation

Lifting this observation to execution sequences is simple: we only have to ensure that a valid sequence remains valid after swapping. This is true since we restricted validity only to liveness. However, more complex assumptions over the environment could link input and output behavior of different devices or their relative speed. As mentioned before, these assumptions would be encoded as properties over the scheduling function sI_{PD} of the simulation theorem between VAMP ISA with devices and VAMP gate-level with devices. In this case the invariance of validity has to be proven before applying the reordering theorem that we present in the next section.

We call the non-interference observation *valid over a set of sequences Seq*, if swapping two non-interfering events at an arbitrary position of a sequence will again result in a sequence of the same set. Non-interfering events are defined in accordance with the observation:

$$\diamond^S(ev_1, ev_2) \equiv \;\; (ev_1 = (id_{D1}, \dots) \wedge ev_2 = (id_{D2}, \dots) \implies id_{D1} \neq id_{D2})$$

We define the swap operator on sequences as follows:

$$swap :: Seq_T \times \mathbb{N} \rightarrow Seq_T$$

$$\begin{aligned} swap(seq, i) = \;\; \lambda x. \quad & \textbf{if } \; x = i \textbf{ then } seq(i+1) \\ & \textbf{else } (\textbf{if } \; x = i+1 \textbf{ then } seq(i) \\ & \qquad \textbf{else } seq(x)) \end{aligned}$$

Next, we define *non-interfering permutation equivalence* of two sequences, denoted with $\sim_\diamond^S$. It states that two sequences are equal up to swapping at a single, non-interfering position:

$$seq \sim_\diamond^S seq' \equiv \exists i. seq(i) \diamond^S seq(i+1) \wedge swap(seq, i) = seq'$$

Now, we can state the notion of observation validity, formulated above, as follows: We call the non-interference observation valid over a set of execution sequences, if this set is closed under non-interfering permutations:

$$valid\text{-}ob(Seq) \equiv \forall seq \in Seq, seq' \,.\, seq \sim_\diamond^S seq' \implies seq' \in Seq$$

Separability

We define the following simple criterion to determine whether the basic observation is valid over a set of execution sequences. Let $\pi(seq, id_D)$ denote the *projection* of a sequence seq to a given device id_D, i.e., it returns the subsequence of external steps of device id_D:

$$\pi :: Seq_T \times \mathfrak{D} \rightarrow Seq_T$$
$$\pi(seq, id_D) \equiv filter(seq, \lambda ev \,.\, (\exists eifi \,.\, ev = (id_D, eifi)))$$

According to Lemma 1 the filter π is well-defined, since we assume liveness of all devices.

A predicate over execution sequences is called *separable* if it can be expressed as a conjunction of predicates over projected execution sequences:

$$separable(Q) \equiv \exists p_0 \dots p_7 \,.\, \forall seq \,.\, Q(seq) = p_0(\pi(seq, 0)) \wedge \dots \wedge p_7(\pi(seq, 7))$$

Lemma 4 (Separable Valid Sequences) *If the set of valid sequences is separable, then the non-interference observation holds over this set:*

$$separable(\lambda seq \,.\, seq \in Seq_V) \implies valid\text{-}ob(Seq_V)$$

Proof. We prove the lemma by contradiction. Given a sequence $seq \in Seq_V$ we claim that a permuted sequence $seq' \sim_{\diamond}^{S} seq$ is not in Seq_V. From separability we conclude, that predicates p_0 to p_7 exist for which we have: $p_0(\pi(seq, 0)) \wedge \cdots \wedge p_7(\pi(seq, 7))$ and $\neg(p_0(\pi(seq, 0)) \wedge \cdots \wedge p_7(\pi(seq, 7)))$. However, projections are invariant under the swapping of non-interfering events, i.e. we have $\pi(seq, i) \equiv \pi(seq', i)$ and hence a contradiction. **q.e.d.**

In the following we use the terms set and characteristic function of a set, interchangeably. It is easy to see that the originally defined set Seq_V is separable. Note that in the formal work we did not apply the general notion of separability. Rather we proved that the valid observation holds on the considered set Seq_V manually.

3.2.2 Reordering

We study when sequential proofs over a given assembly code can be generalized to arbitrary computations. In the remainder of this thesis, we abbreviate the configurations of a processor-local computation by $asm^t = \Delta_{asm}(asm, t)$ and the configurations of a computation of the combined system by $asmd^{seq,t} = fst(\mathit{\Delta}_{\text{asmd}}(asmd, seq, t))$.

In the simplest case the processor does not access any devices. Since interrupts are masked, processor computations yield the same result in the combined model regardless of device steps. This is expressed by the next lemma, which also ensures that correctness of a processor invariant Q is independent of device steps.

Lemma 5 (No Device Access)

$$\begin{aligned} &(\forall t \le T \,.\, da(asm^t) = P \wedge Q(asm^t)) \implies \\ &\quad \forall seq \,.\, (asm^0 = asmd^{seq,0}.\mathsf{proc} \implies \exists T' \,.\;\; asmd^{seq,T'}.\mathsf{proc} = asm^T \wedge \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\forall t' \le T' \,.\, Q(asmd^{seq,t'}.\mathsf{proc}))) \end{aligned}$$

The proof is conducted by induction on t and is based on processor liveness guaranteed by the set of valid execution sequences.

The stated lemma is useful in two situations. First, it allows to reason locally about local steps in the execution of a device driver. Second, it is applicable when reasoning about code of a high-level programming language without direct device access. In this case, code correctness proofs of the code in the high-level language can be performed purely sequentially (see Section 3.3.7).

In a more general case the processor accesses only a certain device id_D. We call such parts of the computation *pure*. This is our assumption for drivers;

it can usually be shown statically or for local processor computations. Furthermore, we define device configurations to be *stable* if they do not change under external transitions. These predicates are defined formally as follows:

$$\mathit{pure} :: C_{\text{asmd}} \times \mathit{Seq}_T \times \mathbb{N} \times \mathfrak{D} \to \mathit{bool}$$
$$\mathit{pure}(\mathit{asmd}^0, \mathit{seq}, T, \mathit{id}_D) \equiv \forall t < T \,.\, \mathit{da}(\mathit{asmd}^{\mathit{seq},t}.\mathsf{proc}) \in \{P, \mathit{id}_D\}$$

$$\mathit{stable} :: S_{\mathfrak{D}} \to \mathit{bool}$$
$$\mathit{stable}(c_X) \equiv \forall \mathit{eifi} \,.\, \delta_{\mathit{dsty}(c_X)}(c_X, \mathit{eifi}, \mathit{mifi}_\epsilon) = c_X$$

The *empty sequence* is the schedule where only the processor takes steps. It is defined as $\mathit{emp}(t) = P$ for all t. In pure computations where the accessed device is stable, sequential properties proven over the empty sequence can be generalized to properties over arbitrary sequences.

Lemma 6 (Pure Sequences and Stable Devices)

$$\begin{array}{l}
\mathit{pure}(\mathit{asmd}^0, \mathit{emp}, T, \mathit{id}_D) \wedge (\forall t < T \,.\, \mathit{stable}(\mathit{asmd}^{\mathit{emp},t}.\mathsf{devs}(\mathit{id}_D)))\wedge \\
(\forall t' \leq T \,.\, Q(\mathit{asmd}^{\mathit{emp},t'}.\mathsf{proc})) \implies \\
\quad (\forall \mathit{seq} \,.\, \exists T' \,.\quad \mathit{asmd}^{\mathit{emp},T}.\mathsf{proc} = \mathit{asmd}^{\mathit{seq},T'}.\mathsf{proc} \wedge \\
\qquad\qquad\qquad \mathit{asmd}^{\mathit{emp},T}.\mathsf{devs}(\mathit{id}_D) = \mathit{asmd}^{\mathit{seq},T'}.\mathsf{devs}(\mathit{id}_D)\wedge \\
\qquad\qquad\qquad (\forall t' \leq T' \,.\, Q(\mathit{asmd}^{\mathit{seq},t'}.\mathsf{proc})))
\end{array}$$

The proof is conducted by induction on T and is based on processor and device liveness guaranteed by the set of valid execution sequences.

Note, that stability of a device is a relatively strong assumption, but sufficient for handling a hard disk driver. For other devices, the notion of stability should be refined, requiring stability only for those parts of the device that are accessed by the processor (e.g. the head of a input queue in a serial interface).

In general, of course, driver correctness can not be shown solely using Lemmas 5 and 6. In the situations not covered by these lemmas, we may still assume that only the processor or the device id_D are being scheduled. We call such fragments of an execution sequence *reduced*. Formally, we define

$$\mathit{reduced} :: \mathit{Seq}_T \times \mathbb{N} \times \mathbb{N} \times \mathfrak{D} \to \mathit{bool}$$

$$\mathit{reduced}(\mathit{seq}, T_1, T_2, \mathit{id}_D) \equiv \forall T_1 \leq t < T_2 \,.\, \mathit{seq}(t) = P \vee \mathit{seq}(t) = (\mathit{id}_D, ?)$$

Complementary, a fragment can be *free* of steps of a device id_D or the processor:

$$\mathit{free} :: \mathit{Seq}_T \times \mathbb{N} \times \mathbb{N} \times \mathfrak{D} \to \mathit{bool}$$
$$\mathit{free}(\mathit{seq}, T_1, T_2, \mathit{id}_D) = \forall T_1 \leq t < T_2 \,.\, \mathit{seq}(t) \neq P \wedge \mathit{seq}(t) \neq (\mathit{id}_D, \dots)$$

If we have a separable valid sequence, the theorem below states that a pure computation can always be reordered into a reduced part and followed by a free part. The resulting overall state of both computations are equal.

Theorem 2 (Reordering of Sequences)

$$\begin{aligned}
&\mathit{pure}(\mathit{asmd}^0, \mathit{seq}, T, \mathit{id}_X) \implies \\
&\quad(\exists \mathit{seq}' \in \mathit{Seq}_V, T_m . \quad \mathit{reduced}(\mathit{seq}', 0, T_m, \mathit{id}_X) \wedge \\
&\qquad\qquad\qquad\qquad\quad \mathit{free}(\mathit{seq}', T_m, T, \mathit{id}_X) \wedge \\
&\qquad\qquad\qquad\qquad\quad \mathit{asmd}^{\mathit{seq},T} = \mathit{asmd}^{\mathit{seq}',T})
\end{aligned}$$

Proof. The theorem is proven by induction on T. The induction base is trivially true. In the induction step (variables of the induction hypothesis are marked by a superscript h), $T = T^h + 1$, we have to consider two cases: either the last step of the computation involves some device with $\mathit{id}_Y \neq \mathit{id}_X$ (i.e. processor access to id_Y or external step of id_Y), or not. In the first case, we can infer from the assumption *purity* that $\mathit{fst}(\mathit{seq}(T)) = \mathit{id}_Y$. We instantiate T_m with the corresponding variable of the induction hypothesis T_m^h. seq' is instantiated with a sequence which is equal up to position T to the one obtained from the induction hypothesis: $\lambda x .\ \mathbf{if}\ x < T\ \mathbf{then}\ \mathit{seq}_h(x)\ \mathbf{else}\ \mathit{seq}(x)$. Using these instantiations, correctness follows straightforward: from the induction hypothesis we know that $\mathit{asmd}^{\mathit{seq},T-1} = \mathit{asmd}^{\mathit{seq}',T-1}$ and since $\mathit{seq}(T) = \mathit{seq}'(T)$ also the final configurations are equal.

For the second case, we can infer that only device id_X or the processor may be involved in the last step, i.e. we have $\mathit{Da}(\mathit{asmd}^{\mathit{seq},T-1}, \mathit{seq}(T)) \in \{\{\mathit{id}_X\}, \{P, \mathit{id}_X\}, \{P\}\}$. By applying the induction hypothesis we conclude that for the execution of $T-1$ steps, reordering is correct. Our goal is to show that the event $\mathit{seq}(T)$ can be shifted to the position $T_m^h + 1$ without changing the outcome of the computation. That means, we instantiate T_m with $T_m^h + 1$ and seq' with $\lambda x .\ \mathbf{if}\ x \leq T_m^h\ \mathbf{then}\ \mathit{seq}'_h(x)\ \mathbf{else}\ (\mathbf{if}\ x = T_m^h + 1\ \mathbf{then}\ \mathit{seq}(T)\ \mathbf{else}\ \mathit{seq}'_h(x-1))$. Since Seq_V is separable we can infer by Lemma 4 that $\mathit{seq}' \in \mathit{Seq}_V$. The conclusion can now be shown by repeatedly applying the basic observation. **q.e.d.**

By generalizing this result, the execution of drivers controlling different devices can also be separated, enabling modular verification of device drivers.

In Fig. 3.7 on page 46 we show an example of a complete execution of a driver for some device id_X. By applying Theorem 2 we soundly reorder the execution of any device id_Y after the termination of the driver (top line to middle line). The interaction between the driver and the corresponding device can now be specified by a single atomic state update (middle line to bottom line).

Similarly, one can state a theorem which describes non-interference in a pure computation from the viewpoint of a not accessed device. Its computation can be considered as completely independent.

The function *step-nr* returns the count of events in which a given device is involved:

$$\mathit{step\text{-}nr} :: \mathit{Seq}_T \times \mathfrak{D} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$
\begin{aligned}
\mathit{step\text{-}nr}(\mathit{seq}, \mathit{id}_X, T) = \quad & \textbf{if } T \leq 1 \textbf{ then } 0 \textbf{ else} \\
& \quad (\textbf{if } \mathit{seq}(x) = (\mathit{id}_X, \dots) \textbf{ then} \\
& \qquad \mathit{step\text{-}nr}(\mathit{seq}, \mathit{id}_X, T-1) + 1 \\
& \quad \textbf{else} \\
& \qquad \mathit{step\text{-}nr}(\mathit{seq}, \mathit{id}_X, T-1))
\end{aligned}
$$

Using this function, we can state a non-interfering theorem from the not-involved devices' point of view as follows:

Theorem 3 (Non-interference of devices)

$$
\begin{aligned}
& \mathit{pure}(\mathit{asmd}^0, \mathit{seq}, T, \mathit{id}_X) \implies \\
& \quad (\forall \mathit{id}_Y \neq \mathit{id}_X \,. \\
& \qquad \mathit{asmd}^{\mathit{seq},T}.\mathsf{devs}(\mathit{id}_Y) = \mathit{asmd}^{\pi(\mathit{seq},\mathit{id}_Y),\mathit{step\text{-}nr}(\mathit{seq},\mathit{id}_Y,T)}.\mathsf{devs}(\mathit{id}_Y))
\end{aligned}
$$

The proof of this theorem follows the lines of the proof of Theorem 2. Note, that the conclusion of the theorem is an assumption formalized first for the page-fault handler correctness theorem [Sta09].

3.3 C0 and Devices

The language C0 was designed as a subset of C which is expressive enough to allow implementations of all encountered system code in the Verisoft project, while remaining handy enough for verification. Therefore, we restricted ourselves to a type-safe fragment of C, without pointer arithmetic. However in the context of system-code verification we have to deal with portions of inline assembly code that break the abstraction of structured C0 programs: low-level hardware intrinsics as processor registers, explicit memory model and devices become visible.

The language stack of the Verisoft project comprises three flavors of C0 reasoning: Hoare logic, small-step semantics and an intermediate big-step semantics. Having in mind that devices are executed in parallel with the processor, and that computations of the processor may be interrupted, only a small-step semantics is adequate for verifying drivers and interleaved applications. However, ultimately, the right level to express overall correctness of system software, say the kernel, is VAMP ISA with devices; only there all relevant components, as for example the mode register, become visible.

Still, conducting all the code verification at the level of small step semantics or even below is not intended. Otherwise, one would abdicate the whole power of Hoare logic and the corresponding verification condition generator. The solution is to abstract low-level components by an extended state and to encapsulate the effects of inline assembly code by so called XCalls, which are atomic specifications manipulating both the extended state and the original C0 machine. First, by enriching the semantics stack with XCalls, we lift assembly code and driver semantics into Hoare logic. Then, by proving implementation correctness of XCalls we transfer results proven in Hoare logic down to VAMP assembly with devices.

In this section we detail the syntax and semantics of C0 small-step semantics, summarize C0 compiler correctness [Lei08], extend it to be applicable on VAMP assembly with devices and show how to deal with dynamic restrictions on the memory consumption. In the subsequent section, we describe how inline assembly portions are embedded into C0 code and how this embedding is extended to device computations. Finally, in the last section of this chapter, the concept of XCalls is introduced by defining an extended C0 small step semantics and a corresponding correctness theorem.

3.3.1 Types

Types are either elementary, composed or pointer types. The set of elementary types is given by $\mathsf{Boolean}$, Int, $\mathsf{Unsigned}$, Char and Null. Composed types are arrays and structs. An array type is defined by its size and its element type, e.g. $\mathsf{Array}(10, \mathsf{Int})$ denotes the type of integer arrays of length 10. Struct types are similar to the record types of Isabelle. They are defined by a set of fields, each field consists of a name and a type. Pointer types in C0 are similar to references in ML, i.e. they are always parametrized over some type. For example $\mathsf{Ptr}(\mathsf{Int})$ denotes the type of pointers to elements of type integer. The null pointer has the elementary type Null.

Given the set *field*$_n$ of field names and a set *type*$_n$ of type names, C0 types are defined by the following abstract data type:

$$
\begin{aligned}
ty ::= \; & \mathsf{Boolean} \mid \\
& \mathsf{Int} \mid \\
& \mathsf{Unsigned} \mid \\
& \mathsf{Char} \mid \\
& \mathsf{Null} \mid \\
& \mathsf{Ptr}\ \mathbf{of}\ type_n \mid \\
& \mathsf{Array}\ \mathbf{of}\ \mathbb{N} \times ty \mid \\
& \mathsf{Struct}\ \mathbf{of}\ (field_n \times ty)\ list
\end{aligned}
$$

The set of all used types of a C0 program is given by its *type environment*, which is a list containing mappings between type names and types:

$$tenv_T = (type_n \times ty)\ list$$

For each type we define a *typesize*. Elementary types and pointer types have size one, whereas the size of arrays and structs are computed recursively:

$$type\text{-}size :: ty \rightarrow \mathbb{N}$$

$$
\begin{array}{lll}
type\text{-}size & (\mathsf{Array}(s,T)) & = type\text{-}size(T) \cdot s \\
& (\mathsf{Struct}(fs)) & = \sum_{i=0}^{length(fs)-1} type\text{-}size(snd(fs!i)) \\
& (else) & = 1
\end{array}
$$

3.3.2 Expressions and Statements

Expressions can be made of literals, i.e. simple values (defined below), variables, array or structure accesses, expressions with common binary or unary operators, pointer dereferencing and the address-of operator.

Let the set *var*$_n$ denot possible variable names. Moreover, let sets BOP and UOP denote the names of binary and unary C0 operators. Then C0 expressions are defined by the following abstract datatype:

$$
\begin{aligned}
expr ::= \; & \mathsf{Lit}\ \mathbf{of}\ sval \mid \\
& \mathsf{VarAcc}\ \mathbf{of}\ var_n \mid \\
& \mathsf{ArrAcc}\ \mathbf{of}\ expr \times expr \mid \\
& \mathsf{StructAcc}\ \mathbf{of}\ expr \times \mathbb{N} \mid \\
& \mathsf{BinOp}\ \mathbf{of}\ \mathsf{BOP} \times expr \times expr \mid \\
& \mathsf{UnOp}\ \mathbf{of}\ \mathsf{UOP} \times expr \mid \\
& \mathsf{AddrOf}\ \mathbf{of}\ expr \mid \\
& \mathsf{Deref}\ \mathbf{of}\ expr
\end{aligned}
$$

C0 defines all common imperative statements: Skip, statement composition, assignments to variables, memory allocation for pointers of a given type, static function calls and returns from functions, if-then-else statements and loops. Function calls take as parameters, the name of the variable to which the result is written, the function name and a list of parameter expressions.

Given a set of function names fun_n and a set of variable names var_n. Then the set of C0 statements is defined by:

$$
\begin{array}{rl}
\mathit{stmt} ::= & \mathsf{Skip} \mid \\
& \mathsf{Comp}\ \mathbf{of}\ \mathit{stmt} \times \mathit{stmt} \mid \\
& \mathsf{Ass}\ \mathbf{of}\ \mathit{expr} \times \mathit{expr} \mid \\
& \mathsf{PAlloc}\ \mathbf{of}\ \mathit{expr} \times \mathit{type}_n \mid \\
& \mathsf{SCall}\ \mathbf{of}\ \mathit{var}_n \times \mathit{fun}_n \times \mathit{expr\ list} \mid \\
& \mathsf{Return}\ \mathbf{of}\ \mathit{expr} \mid \\
& \mathsf{Ifte}\ \mathbf{of}\ \mathit{expr} \times \mathit{stmt} \times \mathit{stmt} \mid \\
& \mathsf{Loop}\ \mathbf{of}\ \mathit{expr} \times \mathit{stmt}
\end{array}
$$

For convenience we will abbreviate in the following $\mathsf{Comp}(s_1, s_2)$ by $s_1;;s_2$. Moreover, sometimes, we abbreviate function calls $\mathsf{SCall}(x, \mathit{fn}, \mathit{exps})$ by $x = \mathtt{fn}(\mathit{exps}!0, \ldots, \mathit{exps}!n)$ for $n = \mathit{length}(\mathit{exps}) - 1$.

3.3.3 Values and Configuration

Value Representation

The C0 small-step semantics has a very explicit memory model with no structured values. Simple values correspond to the elementary types Boolean, Int, Unsigned, Char and to pointer values. Since C0 does not allow any pointer arithmetic, values of pointers are simply identifiers of variables. Variable identifiers are defined in the next section. In the following we will map elementary C0 types to corresponding types in Isabelle, i.e. the set of simple values is given by the abstract data type:

$$
\begin{array}{rl}
\mathit{sval} ::= & \mathsf{BoolV}\ \mathbf{of}\ \mathit{bool} \mid \\
& \mathsf{IntV}\ \mathbf{of}\ \mathbb{Z} \mid \\
& \mathsf{UnsignedV}\ \mathbf{of}\ \mathbb{N} \mid \\
& \mathsf{CharV}\ \mathbf{of}\ \mathit{char}_T \mid \\
& \mathsf{PtrV}\ \mathbf{of}\ \mathit{var\text{-}id}
\end{array}
$$

Values of composed types are flattened and represented by so called *content functions* of type $\mathit{content}_T = \mathbb{N} \to \mathit{sval}$. For example, an array of integers containing the two elements 1 and 2 is represented by the content function:

$$
\begin{array}{l}
\lambda x\ .\ \mathbf{if}\ \ x = 0\ \mathbf{then}\ 1\ \mathbf{else} \\
\qquad\quad (\mathbf{if}\ \ x = 1\ \mathbf{then}\ 2\ \mathbf{else}\ \mathit{undef})
\end{array}
$$

Memory frame

Memory frames are records storing values of variables. They have type $\mathit{mem\text{-}frame}_T$ with the following record fields:

$\mathsf{vars} \in (\mathit{var}_n \times \mathit{type}_n)\ \mathit{list}$	List of descriptors of variables stored in the memory frame.
$\mathsf{content} \in \mathit{content}_T$	Values of all variables flattened and stored into a single content function.

The offset of the ith variable denotes the position in the content in which the first simple value of the variable is stored. Formally we have:

$$\mathit{off} :: \mathit{tenv}_T \times \mathit{mem\text{-}frame}_T \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\mathit{off}(\mathit{tenv}, \mathit{mf}, i) = \sum_{k=0}^{i-1} \mathit{type\text{-}size}(\mathit{tenv}?\mathit{tn}),$$

where tn denotes the type name of the kth variable, i.e. $\mathit{tn} = \mathit{snd}(\mathit{mf}.\mathsf{vars}!k)$.

We illustrates how values are stored in a memory frame by the following example. Let x be the variable name of the simple integer array defined above, and let i be the list index of x in $\mathit{mf}.\mathsf{vars}$. We compute the offset of the variable x in the memory frame, by:

$$\mathit{off}_{\mathrm{x}} = \mathit{off}(\mathit{tenv}, \mathit{mf}, i)$$

Both values of the array are stored as follows in the memory frame mf:

$$\begin{aligned} \mathit{mf}.\mathsf{content}(\mathit{off}_{\mathrm{x}}) &= \mathsf{IntV}(1) \\ \mathit{mf}.\mathsf{content}(\mathit{off}_{\mathrm{x}} + 1) &= \mathsf{IntV}(2) \end{aligned}$$

Next, we define the size of a memory frame mf as the number of simple values stored in it:

$$\mathit{frame\text{-}size} :: \mathit{tenv}_T \times \mathit{mem\text{-}frame}_T \rightarrow \mathbb{N}$$

$$\mathit{frame\text{-}size}(\mathit{tenv}, \mathit{mf}) = \mathit{off}(\mathit{tenv}, \mathit{mf}, \mathit{length}(\mathit{mf}.\mathsf{vars}))$$

Parameters

Each C0 machine is parametrized over a type environment and a set of function declarations, called *procedure table*. A function declaration is of type $\mathit{fun\text{-}decl}_T$ and has the following record fields:

params $\in (var_n \times type_n)$ *list*	The list of function parameters. A parameter is given by its name and type.
locals $\in (var_n \times type_n)$ *list*	The list of local variable descriptors. A variable descriptor consists of a variable name and type.
ret-type $\in type_n$	The return type of the function.
body $\in stmt$	The function body.

A procedure table is a mapping from function names to function declarations, i.e. it is of type $proctable_T = fun_n \rightarrow fun\text{-}decl_T$.

We define the size of a function declaration as the sum of the type size of the parameters and local variables:

$$fun\text{-}size :: tenv_T \times fun\text{-}decl_T \rightarrow \mathbb{N}$$

$$\begin{aligned} &fun\text{-}size(tenv, fnd) \equiv \\ &\quad \textstyle\sum_{i=0}^{length(fnd.\mathsf{params})-1} type\text{-}size(tenv?(snd(fnd.\mathsf{params}!\ i)))+ \\ &\quad \textstyle\sum_{i=0}^{length(fnd.\mathsf{locals})-1} type\text{-}size(tenv?(snd(fnd.\mathsf{locals}!\ i))) \end{aligned}$$

Note, that the frame size is equal to the size of the corresponding function declaration.

Configuration

The memory configuration of a C0 machine has record type $memC0_T$ and consists of three components:

gm $\in mem\text{-}frame_T$	The global memory.
lms $\in (mem\text{-}frame_T \times var\text{-}id)$ *list*	The local memory stack. For each not yet returned function call, it contains a memory frame and a variable identifier to which the result is written.
hm $\in mem\text{-}frame_T$	The heap memory. Note that variables in the heap are nameless, i.e. we ignore the name component.

Now, we can define Configurations of C0 machines as records of type C_{co} with the following fields:

prog $\in stmt$	The current statement to execute.
mf $\in memC0_T$	The memory configuration.

We call the number of local memory frames in the current memory configuration of a C0 machine *recursion depth*. The recursion depth is denoted by the function $rd(mf) \equiv length(mf.\mathsf{lms})$. The frame of the currently executed function call is the top most one, i.e. $mf!(rd(mf)-1)$.

Sometimes, it is helpful to have the description of the C0 machine as a whole, i.e. consisting of configuration and parameters. We call such a description monolithic C0 configuration. It has the following record type:

$$C_{\mathrm{com}} = (\mathsf{tenv} : tenv_T, \mathsf{pt} : proctable_T, \mathsf{conf} : C_{\mathrm{co}})$$

3.3.4 Variables and Evaluation

Variable Identifiers

A global variable is a variable allocated in the global memory frame. It is identified solely by a name. A local variable is a variable residing in one of the local memory frames. It is identified by a name and the number of the stack frame, in which it is stored. Identifiers of heap variables consists only of their number in the heap frame.

For variables of structured type, we introduce the notion of subvariables. A subvariable is either an array element or a structure field. It is identified by its *root* variable and either an index or a field name:

$$\begin{aligned} var\text{-}id ::= \;& \mathsf{GV}\ \textbf{of}\ var_n \mid \\ & \mathsf{LV}\ \textbf{of}\ \mathbb{N} \times var_n \mid \\ & \mathsf{HV}\ \textbf{of}\ \mathbb{N} \mid \\ & \mathsf{ArrV}\ \textbf{of}\ var\text{-}id \times \mathbb{N} \mid \\ & \mathsf{StrV}\ \textbf{of}\ var\text{-}id \times field_n \end{aligned}$$

Examples: $\mathsf{GV}(vn)$ is a variable with name vn residing in the global memory, $\mathsf{LV}(rd-1, vn)$ is a local variable with the same name allocated in the top most frame (if rd denotes the current recursion depth), and the ith variable on the heap is identified by $\mathsf{HV}(i)$.

The predicate *is-elementary* indicates for a given configuration and a variable identifier whether the latter is defined as subvariable or not. Given a configuration c we denote the set of all valid variable identifiers by *valid-vars*(c). For a formal definition of *valid-vars* we refer to [Lei08].

Evaluation

Variables and expressions evaluate to records of type $dataslice_T$, consisting of two fields:

$\mathsf{type} :: type_n$	The type of the value.
$\mathsf{content} :: \mathbb{N} \rightarrow sval$	The content storing the flattened value.

Note, that the content of a given data slice ds contains relevant information only for the size of its type, i.e. from position 0 to position $type\text{-}size(tenv?(ds.\mathsf{type}))$.

The function $vlookup :: C_{\text{co}} \times var\text{-}id \rightarrow dataslice_T$ returns for a given C0 state and a variable identifier the corresponding value in form of a data slice. We outline its definition with the following example: Let $x = \mathsf{GV}(vn)$ be a global variable of type ty, defined in the C0 configuration c. Moreover let gm_{c} abbreviate access to global memory, i.e. $gm_{\text{c}} = c.\mathsf{mf}.\mathsf{gm}$. A value lookup would return a data slice with the following content:

$$vlookup(c, x).\mathsf{content}(i) = \\ \begin{cases} gm_{\text{c}}.\mathsf{content}(\mathit{off}(tenv, gm, vn) + i) & : \ i < type\text{-}size(ty) \\ arbitrary & \textbf{otherwise} \end{cases}$$

The function $eval :: C_{\text{co}} \times expr \rightarrow dataslice_T$ evaluates C0 expressions to data slices or to the value $undef$. Pointers (expressions of type AddrOf) evaluate to variable identifiers. For variable accesses, it is first determined whether there is a matching variable definition either in the global, or top most memory frame. In this case the result is determined by the function $vlookup$, otherwise $undef$ is returned. Moreover, we use an update function $upd_{\text{mem}} :: C_{\text{co}} \times var_n \times dataslice_T$ which updates the value of a variable in the given memory. The following relation between evaluation and updates should hold: $eval(upd_{\text{mem}}(c, vn, ds), \mathsf{VarAcc}(vn)) = ds$. Formal definitions of expression evaluation and variable update can be found in [Lei08].

Note, that equality of the values of two expressions is not given by equality on the result of evaluation, since not each position in the content of a data slice contains relevant data (the relevant positions are determined by the size of the value type).

3.3.5 Small Step Semantics

The small-step semantics of C0 is defined by the transition function:

$$\delta_{c0} :: C_{\text{co}} \times tenv_T \times proctable_T \rightarrow (C_{\text{co}}\ option)$$

As input it takes a description of the C0 machine, consisting of the configuration, the type environment and the procedure table. It returns either $\perp$, in case the computation got stuck due to an error, or in the successful case it returns a new C0 configuration. Computations can for example get stuck due to the evaluation of illegal expressions.[4]

The execution of i many consecutive C0 transitions is defined by the function:

$$\Delta_{c0} :: C_{\text{co}} \times tenv_T \times proctable_T \times \mathbb{N} \rightarrow (C_{\text{co}}\ option)$$

[4]Indeed the transition function also requires an additional predicate over the heap memory, which indicates whether all available heap space is consumed so far, or not. Due to garbage collection, this predicate is not fixed at compile time. For simplicity, in the following, we ignore garbage collection and hence omit the new parameter.

$$\begin{aligned} \Delta_{c0} \quad & (c, tenv, pt, 0) &&= \lfloor c \rfloor \\ & (c, tenv, pt, i+1) &&= \begin{cases} \Delta_{c0}(c', tenv, pt, i) & : \delta_{c0}(c, tenv, pt) = \lfloor c' \rfloor \\ \bot & \textbf{otherwise} \end{cases} \end{aligned}$$

In the following we abbreviate $\Delta_{c0}(c, tenv, pt, n)$ by ${\delta_{c0}}^n(c, tenv, pt)$.

The transition function δ_{c0} is recursively defined on the structure of the program. In the remainder of this work our main interest focuses on the execution of function calls, since they are later on used to encapsulate the assembly driver. Therefore, in the next paragraphs, we describe the semantics of composed statements, function calls and function returns.

Statement Composition

Let the program rest be a statement composition, i.e. $c.\mathsf{prog} = s_1;;s_2$. If the first statement is Skip, it is simply deleted and the memory configuration stays unchanged. Thus, in case of $s_1 = \mathsf{Skip}$, we have $c' = (\mathsf{prog} = s_2, \mathsf{mf} = c.\mathsf{mf})$.

Otherwise, the first statement is recursively executed. If this execution is stuck, we return $\bot$. If not, we take the result of the recursive call and delete the first statement:

$$c' = \mathsf{the}(\delta(c[\mathsf{prog} := s_1], tenv, pt))[\mathsf{prog} := s_2]$$

Function Call

The semantics of a function is in a nutshell as follows: a new stack frame is created, the parameters are evaluated and copied to this frame and the function call is substituted in the program rest by the body of the called function. It follows a detailed description.

Let the program rest be a function call, i.e. $c.\mathsf{prog} = \mathsf{SCall}(vn, fn, exps)$. Furthermore, let fd_{fn} denote the procedure table entry corresponding to the given function name, x the identifier of the variable corresponding to vn, and $ds!i$ the result of the evaluation of the ith parameter:

$$\begin{aligned} fd_{\text{fn}} &= pt(fn) \\ x &= \begin{cases} \mathsf{LV}(rd(c.\mathsf{mf}) - 1, vn) & : \mathsf{LV}(rd(c.\mathsf{mf}) - 1, vn) \in valid\text{-}vars(c) \\ \mathsf{GV}(vn) & : \mathsf{GV}(vn) \in valid\text{-}vars(c) \\ undef & \textbf{otherwise} \end{cases} \\ ds!i &= eval(c, exps!i) \end{aligned}$$

The transition is non stuck if and only if: (i) A corresponding procedure table entry exists, (ii) the return variable is valid, i.e. it is either a global variable or a variable of the top most local memory frame, (iii) all parameters evaluate to some defined value. Formally, these assumptions are given by:

$$fd_{\text{fn}} \neq undef \wedge x \neq undef \wedge ds!i \neq undef$$

If one of the assumption is violated, the transition returns with $\bot$. Otherwise, a new frame is created according to the declaration of the function in the procedure table:

- The list of variables is defined as concatenation of parameters and local variables defined in the corresponding function declaration:
$$\mathit{frm}_{\mathrm{fn}}.\mathsf{vars} = \mathit{fd}_{\mathrm{fn}}.\mathsf{params}@\mathit{fd}_{\mathrm{fn}}.\mathsf{locals}$$
- The content of the new frame is defined as concatenation of the contents of all parameters:
$$\begin{aligned}&\forall i < \mathit{length}(\mathit{fd}_{\mathrm{fn}}.\mathsf{params}).\\ &\quad (\forall k < \mathit{off}(\mathit{tenv}, \mathit{frm}_{\mathrm{fn}}, i+1).\\ &\qquad \mathit{off}(\mathit{tenv}, \mathit{frm}_{\mathrm{fn}}, i) \le k\\ &\qquad \Longrightarrow \mathit{frm}_{\mathrm{fn}}.\mathsf{content}(k) = (\mathit{ds}!i).\mathsf{content}(k - \mathit{off}(\mathit{tenv}, \mathit{frm}_{\mathrm{fn}}, i)))\end{aligned}$$

Finally, the new configuration c' results in:

$$\begin{aligned}c' = (\ &\mathsf{prog} = \mathit{fd}_{\mathrm{fn}}.\mathsf{body},\\ &\mathsf{mf} = c.\mathsf{mf}[\mathsf{lms} := (\mathit{frm}_{\mathrm{fn}}, x)\#c.\mathsf{mf}.\mathsf{lms}])\end{aligned}$$

Function Return

In a nutshell, the return statement is executed as follows: (i) The new program rest is set to Skip, (ii) the top most frame is deleted, and (iii) the result variable is updated with the evaluated return expression. It follows a detailed description.

Let the program rest denote a return from a function call, i.e. $c.\mathsf{prog} = \mathsf{Return}(\mathit{exp})$. Furthermore, let $\mathit{var}_{\mathrm{ret}}$ denote the identifier of the variable stored in the top most local stack frame and $\mathit{ds}_{\mathrm{ret}}$ be the result of the evaluation of the given expression to return:

$$\begin{aligned}\mathit{var}_{\mathrm{ret}} &= \mathit{snd}(c.\mathsf{mf}.\mathsf{lms}!(\mathit{rd}(c.\mathsf{mf}) - 1))\\ \mathit{ds}_{\mathrm{ret}} &= \mathit{eval}(c, \mathit{exp})\end{aligned}$$

The transition is non stuck if and only if: (i) The given expression evaluates to a valid value, (ii) the top most frame stores a valid identifier of the variable for storing the return value, i.e. it is either globally defined or in the top most frame. Formally these assumptions are given by:

$$\begin{aligned}&\mathit{ds}_{\mathrm{ret}} \neq \mathit{undef}\\ \wedge\ &\mathit{var}_{\mathrm{ret}} \in \mathit{valid\text{-}vars}(c)\\ \wedge\ &(\exists \mathit{vn}\ .\ \mathit{var}_{\mathrm{ret}} = \mathsf{LV}(\mathit{rd}(c.\mathsf{mf}) - 1, \mathit{vn}) \vee \mathit{var}_{\mathrm{ret}} = \mathsf{GV}(\mathit{vn}))\end{aligned}$$

If one of the assumption is violated, the transition returns with $\bot$. Otherwise, the next configuration is computed as follows:

$$\begin{aligned}c' = \mathit{upd}_{\mathrm{mem}}((\ &\mathsf{prog} = \mathsf{Skip},\\ &\mathsf{mf} = c.\mathsf{mf}[\mathsf{lms} := \mathit{tl}(c.\mathsf{mf}.\mathsf{lms})]),\ \mathit{var}_{\mathrm{ret}}, \mathit{ds}_{\mathrm{ret}})\end{aligned}$$

3.3.6 Useful Properties

In this section we introduce a series of useful properties over C0 programs.

Flattening the Program Rest

The recursive definition of statement composition leads to a tree structure of the code. This becomes extremely cumbersome when arguing about statement equivalence. Consider for example the two statements $(\mathsf{Skip};;\mathsf{Skip});;\mathsf{Skip}$ and $\mathsf{Skip};;(\mathsf{Skip};;\mathsf{Skip})$. Intuitively, both are equivalent, even though they are syntactically different.

We define the function *s2l* which *flattens* the code by simply generating a list of statements:

$$s2l :: stmt \rightarrow stmt\ list$$

$$\begin{array}{lll} s2l & (s_1;;s_2) & = s2l(s_1)@s2l(s_2) \\ & (s) & = [s] \end{array}$$

Note that *s2l* does not define semantical equivalence of statements (i.e. equivalence under execution). Such a definition would require to analyze not only the top-level composition structure of the code, but even to flatten the code within all sub statements.

Using the flattening function, we can define the head of a statement s simply by $hd(s2l(s))$. The following lemma ensures equivalence of execution under the same head of the program rest:

Lemma 7 (Program Header Equivalence) *Let c_1 and c_2 be two C0 configurations with the same memory configuration and the same head of the program rest. Transitions of both configurations will lead again to the same memory configuration.*

$$\begin{array}{ll} & hd(s2l(c_1.\mathsf{prog})) = hd(s2l(c_2.\mathsf{prog})) \\ \wedge & c_1.\mathsf{mf} = c_2.\mathsf{mf} \\ \wedge & \delta_{c0}(c_1, tenv, pt) = \lfloor c_1' \rfloor \\ \Longrightarrow & (\exists c_2' \,.\, \delta_{c0}(c_2, tenv, pt) = \lfloor c_2' \rfloor \\ \wedge & \qquad c_2'.\mathsf{mf} = c_1'.\mathsf{mf}) \end{array}$$

This lemma is proven by structural induction on the program rest.

Dynamic Properties of the Code

Often, we need assumptions which have to hold for the head of the program rest in each step of the execution. Such assumptions may for example require a program never to allocate memory or not to contain nested function calls. Fortunately, we do not need to discharge such conditions dynamically, i.e. at each step of a computation. Rather, a statical analysis of the procedure table and the initial program rest suffices.

For that, we first recursively define the notion of validity of a predicate over a statement as follows:

$$\mathit{valid\text{-}prop} :: \mathit{stmt} \times (\mathit{stmt} \rightarrow \mathit{bool}) \rightarrow \mathit{bool}$$

$$\begin{array}{lll}
\mathit{valid\text{-}prop} & (\mathsf{Skip}, Q) & = Q(\mathsf{Skip}) \\
 & (s_1;; s_2, Q) & = \mathit{valid\text{-}prop}(s_1, Q) \wedge \mathit{valid\text{-}prop}(s_2, Q) \\
 & (\mathsf{Ifte}(e, s_1, s_2), Q) & = Q(\mathsf{Ifte}(e, s_1, s_2)) \wedge \mathit{valid\text{-}prop}(s_1, Q) \wedge \\
 & & \quad \mathit{valid\text{-}prop}(s_2, Q) \\
 & (\mathsf{Loop}(e, s), Q) & = Q(\mathsf{Loop}(e, s)) \wedge \mathit{valid\text{-}prop}(s, Q) \\
 & (\mathit{else}, Q) & = Q(\mathit{else})
\end{array}$$

Once we have proven a property to hold for a statement, we can also infer this property for the head of the statement by the following lemma:

Lemma 8 (Valid Property on Program Head) *Properties which are valid for a statement, are also valid for the head of the statement.*

$$\mathit{valid\text{-}prop}(s, Q) \implies Q(\mathit{hd}(\mathit{s2l}(s)))$$

Next, we extend the valid property predicate to range over all bodies of functions declared in a given procedure table:

$$\mathit{valid\text{-}prop\text{-}pt} :: \mathit{proctable}_T \times (\mathit{stmt} \rightarrow \mathit{bool}) \rightarrow \mathit{bool}$$

$$\mathit{valid\text{-}prop\text{-}pt}(\mathit{pt}, Q) = \forall \mathit{fd} \in \mathit{range}(\mathit{pt}).\ \mathit{valid\text{-}prop}(\mathit{fd}.\mathsf{body}, Q)$$

Now, the next lemma ensures that a property Q is valid over the program rest during the whole execution, if Q was valid over the initial program rest and over the procedure table:

Lemma 9 (Valid Property Invariant) *Given a non stuck computation $\delta^i_{C0}(c, \mathit{tenv}, \mathit{pt}) = \lfloor c' \rfloor$, then the following holds:*

$$\mathit{valid\text{-}prop\text{-}pt}(\mathit{pt}, Q)\ \wedge \mathit{valid\text{-}prop}(c.\mathsf{prog}, Q) \implies \mathit{valid\text{-}prop}(c'.\mathsf{prog}, Q)$$

This lemma is proven by structural induction on the program rest.

Dynamic Properties on parts of the code

Lemma 9 is too restrictive: It requires *all* function bodies to fulfill the demanded property. However, it would suffice only to consider the bodies of those functions which are (possibly) called in the given statement. This applies, for example, if we want to prove that during the execution of a given function call no new memory is allocated, whereas other functions of the program may allocate new memory.

For stating such a stronger lemma, we need to collect all functions that are possibly called during the execution of a given statement. First we define the function *top-scalls*, which returns for a given statement the set of the function names, which appear on the top-level structure of the program rest. Thus, nested calls are ignored.

$$\textit{top-scalls} :: \textit{stmt} \rightarrow (\textit{fun}_n\ \textit{set})$$

$$\begin{array}{lll}
\textit{top-scalls} & (s_1;;s_2) & = \textit{top-scalls}(s_1) \cup \textit{top-scalls}(s_2) \\
 & (\mathsf{Ifte}(e, s_1, s_2)) & = \textit{top-scalls}(s_1) \cup \textit{top-scalls}(s_2) \\
 & (\mathsf{Loop}(e, s)) & = \textit{top-scalls}(s) \\
 & (\mathsf{SCall}(vn, fn, exps)) & = \{fn\} \\
 & (\textit{else}) & = \{\}
\end{array}$$

Using *top-scalls* we can inductively define the set *SCalls*. Parametrized over a procedure table and a set of names of *root* functions, it contains the names of all functions appearing — either at the top-level or nested somewhere deeper — in the body of one of the *root* functions:

$$\textit{SCalls} :: \textit{proctable}_T \times (\textit{fun}_n\ \textit{set}) \rightarrow (\textit{fun}_n\ \textit{set})$$

$$\frac{fn \in root}{fn \in SCalls(pt, root)}$$

$$\frac{fn' \in SCalls(pt, root) \qquad pt(fn') \neq undef \qquad fn \in \textit{top-scalls}(pt(fn').\mathsf{body})}{fn \in SCalls(pt, root)}$$

Our first goal is to prove that the set of function names appearing in the program rest is monotonically decreasing in each transition. We start with a simple observation: *SCalls* is monotone on its second argument.

Lemma 10 (SCalls Monotonicity)

$$X \subseteq Y \implies SCalls(pt, X) \subseteq SCalls(pt, Y)$$

Moreover the application of SCalls always returns a fixpoint:

Lemma 11 (SCalls Fixpoint)

$$SCalls(pt, SCalls(pt, X)) \subseteq SCalls(pt, X)$$

Next we show that the set of top-level functions of a program rest gets smaller in each transition.

Lemma 12 (Monotonicity of top-scalls on transitions) *Given a non stuck computation $\delta_{c0}(c, tenv, pt) = \lfloor c' \rfloor$, then the following holds:*

$$\textit{top-scalls}(c'.\mathsf{prog}) \subseteq SCalls(pt, \textit{top-scalls}(c.\mathsf{prog}))$$

This lemma is proven by structural induction on the program rest.

Now we can prove our goal from above: the set of the function called in the program rest decreases each step.

Lemma 13 (Monotonicity of SCalls on Transitions) *Given a non stuck computation $\delta_{c0}(c, tenv, pt) = \lfloor c' \rfloor$, then the following holds:*

$$SCalls(pt, top\text{-}scalls(c'.\mathsf{prog})) \subseteq SCalls(pt, top\text{-}scalls(c.\mathsf{prog}))$$

This lemma is proven by a simple application of Lemma 12, Lemma 11 and Lemma 10.

Using the last lemma we can prove invariants of the code as follows: we show that the invariant holds on the program rest and on the bodies of all functions appearing in the program rest.

We abbreviate these assumption by the following predicate:

$$\begin{aligned}
&valid\text{-}prop_{\mathsf{code}}(prog, pt, Q) \equiv \\
&\quad (\forall fn \in SCalls(pt, top\text{-}scalls(prog)). \\
&\qquad pt(fn) \neq undef \implies valid\text{-}prop((pt(fn)).\mathsf{body}, Q)) \\
&\wedge \quad valid\text{-}prop(prog, Q)
\end{aligned}$$

The lemma now reads as follows:

Lemma 14 (Valid Property Invariant 2) *Given a non stuck computation $\delta^i_{C0}(c, tenv, pt) = \lfloor c' \rfloor$, then the following holds:*

$$valid\text{-}prop_{code}(c.\mathsf{prog}, pt, Q) \implies valid\text{-}prop(c'.\mathsf{prog}, Q)$$

This lemma is proven by induction on i. We prove the induction step by structural induction on the program rest $c.\mathsf{prog}$. Then we discharge the first assumption of the induction hypothesis by applying Lemma 13.

Note, that the definitions *SCalls* and *top-scalls* have been originally introduced by N. Schirmer, where the stated properties have been proven by the author of this thesis.

3.3.7 Compiler Correctness

The compiler correctness theorem relates C0 computations to VAMP assembly computations of the compiled code. In this section we state the theorem and extend it to VAMP assembly with devices computations.

First, we outline the memory layout of the compiled code on the target machine, i.e. we show how the code, the stack and the heap are allocated. In the C0 small-step semantics, no space requirements for storing the stack are considered. Naturally, things are different on the target machine, which is restricted on free space. Hence, compiler correctness can only be shown under certain memory constraints, which are detailed subsequently. Fortunately, these restrictions may be stated (almost) completely on the level of C0 without resorting to the compilation. Next, we sketch a simulation relation which relates the C0 program, variable values, current statement and pointers to a corresponding assembly state. Then compiler correctness is stated. For the

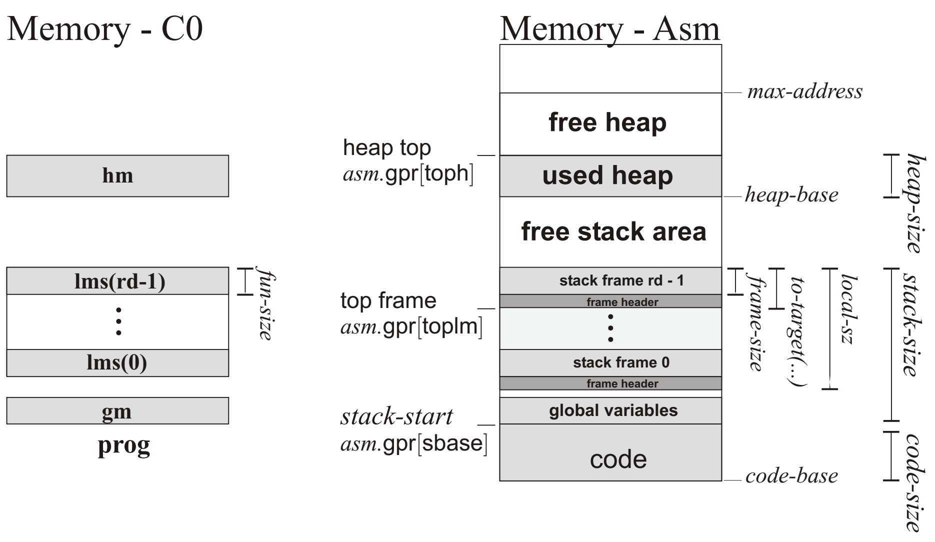


Figure 3.8: C0 memory model and Memory layout on target machine

extension to devices, we prove that the compiled code never accesses devices and that devices do not interfere with compiler correctness. Essentially, the extended compiler correctness theorem enables us to generalize any property proven in the ordinary C0 semantics to arbitrary execution sequences on the target machine with devices.

Memory Layout of Compilation

We first outline the memory layout of compiled programs (cf. Figure 3.8). In short there are three memory regions allocated: (i) A code region, consisting of the compiled code, (ii) the stack, in which each local memory frame is stored and (iii) the heap. For each local memory frame the stack stores the content of its C0 counterpart and additionally a frame header with the address to which the result is written back and the return address of the call in the code.

Parameters of the compiler correctness theorem related to memory allocation are the start address of the code, denoted by *code-base* and the start address of the heap, denoted by *heap-base*. The addresses of the stack start, of the current top element of the heap and of the top most frame are stored at registers with the addresses sbase, toph and toplm.

On Memory Consumption

The goal is to specify the restrictions on memory consumption for the target machine, i.e. VAMP assembly, at the level of C0.

The start address of the stack in the memory is calculated by the *code-base* and the code size. The code size depends on the procedure table, the type environment and the types of the global variables defined by the global memory frame. Where the necessity of the first two parameters is obvious, the latter has a more complicated reason: the code generated for comparing unsigned and signed integer values differ (the first is done by a single hardware instruction whereas for the second a small assembly program is generated). A detailed description of *code-size* can be found in [Lei08].

$$\textit{stack-start}(\textit{tenv}, \textit{pt}, \textit{gms}) = \textit{code-base} + \textit{code-size}(\textit{tenv}, \textit{pt}, \textit{gm})$$

To define the size of the stack, we first have to define the memory consumption of a single function frame on the stack. We already introduced the notion of function size and frame size in the context of C0 function declarations (cf. Section 3.3.3). These definitions differ from their counterpart on the target machine only by the size of a constant frame header FHD. We define the function *to-target* to convert between the size in C0 and the corresponding memory consumption in words on the target machine by $\textit{to-target}(s) = s + \mathsf{FHD}$ Note, that this definition is correct, since the C0 compiler uses a complete word to store a single boolean value or character.

The stack size computes as the sum of the size of the local memory frames and the global memory frame.

The size of all local memory frames is computed by:

$$\textit{local-sz} :: (\textit{mem-frame}_T \times \textit{var-id})\ \textit{list} \rightarrow \mathbb{N}$$

$$\begin{aligned}&\textit{local-sz}(\textit{lms}) \equiv\\&\textstyle\sum_{i=0}^{\textit{length}(\textit{lms})-1} \textit{to-target}(\textit{frame-size}(\textit{tenv}, \textit{lms}!i))\end{aligned}$$

The size of the stack is computed by:

$$\textit{stack-size} :: \textit{tenv}_T \rightarrow C_{\mathrm{co}} \rightarrow \mathbb{N}$$

$$\begin{aligned}&\textit{stack-size}(\textit{tenv}, c) \equiv\\&\quad \textit{to-target}(\textit{frame-size}(\textit{tenv}, c.\mathsf{mf.gm}))\\&\quad +\textit{local-sz}(c.\mathsf{mf.lms})\end{aligned}$$

Note, that after a function call, the stack grows by the size of the declaration of the invoked function:

Lemma 15 (Increase of stack size) *Given a non stuck computation $\delta_{c0}(c, \textit{tenv}, \textit{pt}) = \lfloor c' \rfloor$, then the following holds:*

$$\begin{aligned}&\quad\ \textit{hd}(\textit{s2l}(c.\mathsf{prog})) = \mathsf{SCall}(\textit{vn}, \textit{fn}, \textit{pl})\\\Longrightarrow&\quad\ \textit{stack-size}(\textit{tenv}, c') =\\&\qquad\quad \textit{stack-size}(\textit{tenv}, c) + \textit{to-target}(\textit{fun-size}(\textit{tenv}, \textit{pt}(\textit{fn})))\end{aligned}$$

The space in which the stack is permitted to grow is restricted by the compiler parameter *heap-base*, denoting the start address of the heap. The predicate *frames-in-memory* indicates for a current configuration whether the stack reached this barrier or not:

$$\begin{aligned}&\textit{frames-in-memory}(c, \textit{tenv}, \textit{pt}) =\\ &\textit{stack-start}(\textit{tenv}, c.\mathsf{mf}.\mathsf{gm}) + \textit{stack-size}(\textit{tenv}, c) \leq \textit{heap-base}\end{aligned}$$

Similarly, the heap is restricted in size by the maximum address *max-address* which can be allocated for the C0 machine. Note, that *max-address* is a parameter of the compiler correctness theorem stated above. Given this maximum address and the current size of the heap, *heap-size*(c, *tenv*), the predicate *sufficient-heap* indicates whether the heap reached that barrier or not:

$$\begin{aligned}&\textit{sufficient-heap}(c, \textit{tenv}, \textit{max-address}) =\\ &\textit{heap-base} + \textit{heap-size}(c, \textit{tenv}) < \textit{max-address}\end{aligned}$$

Now we can specify for a given C0 configuration whether its compilation fits the space requirements imposed by the target machine:

$$\begin{aligned}&\textit{sufficient-memory}(\textit{tenv}, \textit{pt}, c, \textit{max-address}) \equiv\\ &\quad\ \textit{frames-in-memory}(c, \textit{tenv}, \textit{pt})\\ &\wedge\ \ \textit{sufficient-heap}(c, \textit{tenv}, \textit{max-address})\end{aligned}$$

Simulation Relation

Consistency between C0 and assembly configurations is defined by means of a simulation relation, which is parametrized over an allocation function. The allocation function maps variable identifiers to their allocated base addresses on the VAMP assembly memory: $\textit{alloc}_T = \textit{var-id} \rightarrow \mathbb{N}$.

During execution the allocation function may change due to

- the execution of an PAlloc statement,
- function calls and returns changing the frame stack,
- garbage collection changing the allocated base address of variables on the heap.

Note that in case no garbage collection is running, the allocation function can be computed from the *code-base* and the current configuration.

The simulation relation *consis* relates the description of a C0 machine — consisting of the type environment, the procedure table and the current configuration — via an allocation function to a VAMP assembly configuration:

$$\textit{consis} :: \textit{alloc}_T \times \textit{tenv}_T \times \textit{proctable}_T \times C_{\mathrm{co}} \times C_{\textit{asm}} \rightarrow \textit{bool}$$

In the following we only sketch its definition (for a formal description see [Lei08]). The predicate *consis*(*alloc*, *tenv*, *pt*, c, *asm*) holds, if the following five conditions are satisfied:

- **Value consistency**. Value consistency is fulfilled if each elementary variable x in c has the same value as the one stored in the physical memory of the assembly state.
- **Pointer Consistency**. Pointer consistency establishes a subgraph isomorphism between the reachable portions of the heap of the C0 and the assembly machine. Let variable x evaluate to some pointer, i.e. the following holds $vlookup(c.\mathsf{mf}, x).\mathsf{content}(0) = \mathsf{PtrV}(y)$. We require that the value stored at the allocated base address of x is the allocated base address of y.
- **Control Consistency**. Control consistency relates the program counters to the current program rest. It requires the dpc to point to the compiled code of the head statement of the program rest.
- **Code Consistency**. For code consistency we require that the compiled C0 program always stays unchanged in the memory of the VAMP assembly machine. The start address of the code is denoted by the compiler parameter *code-base*.
- **Stack Consistency**. This property requires that the stack frame being pointed to in register toplm is consistent with the top most frame in the local memory of the C0 machine. Similar assumptions are made also on the address stored in toph.

Compiler Correctness Theorem

Before stating the simulation theorem, we describe the important conditions:

- Validity of assembly configuration. Compiler correctness is only applicable for valid assembly start configurations. Validity of assembly configurations is subsumed by the predicate $\textit{isa-asm-precond}_{\text{init}}$.
- Validity of C0 configuration. Compiler correctness is only applicable for valid configurations, denoted by the following predicate over C0 machine descriptions:

  $$\textit{valid-C0} :: \textit{tenv}_T \times \textit{proctable}_T \times C_{\text{co}} \rightarrow \textit{bool}$$

  For validity we require among others, that: (i) the code is well-formed, e.g. only declared functions are invoked; the last statement of a function body is a return statement, etc., (ii) the memory is well-typed, e.g. the values of all elementary values in the memory contents are *in range*; for integers and naturals *in range* is defined by the already introduced predicates *asm-int* and *asm-nat*.

- Restrictions on Memory Consumption. Another important condition concerns space: compiler correctness only holds in case enough memory is available on the target machine. The notion of enough memory has been defined in the previous section by the predicate *sufficient-memory* which has to hold in each step of the C0 computation.

- Preconditions of ISA-asm transfer. The assembly model is only an abstraction (and simplification) of an underlying VAMP ISA. For translating computations from VAMP assembly with devices to VAMP ISA with devices via Theorem 1, we need to discharge a series of assumptions on the initial state and assumptions on each step of the execution. These are given by the already defined predicates $\textit{isa-asm-precond}_{\text{init}}$ and $\textit{isa-asm-precond}^{p}_{\text{dyn}}$ (cf. Section 3.1.5).

The correctness theorem is formulated as an N step to T step simulation between C0 and the corresponding VAMP assembly machine:

Theorem 4 (Compiler Correctness)

$$\begin{array}{ll}
 & \textit{consis}(\textit{alloc}, \textit{tenv}, \textit{pt}, c, \textit{asm}) \\
\wedge & \textit{valid-c0}(\textit{tenv}, \textit{pt}, c) \\
\wedge & \textit{isa-asm-precond}_{\textit{init}}(\textit{asm}, \textit{code-base}, \textit{code-size}(\textit{tenv}, \textit{pt}, c.\mathsf{mf}.\mathsf{gm})) \\
\wedge & (\forall i < N.\ \textit{sufficient-memory}(\textit{tenv}, \textit{pt}, \mathsf{the}(\delta^{i}_{C0}(c, \textit{tenv}, \textit{pt})), \textit{max-address})) \\
\wedge & \delta^{N}_{C0}(c, \textit{tenv}, \textit{pt}) = \lfloor c' \rfloor \\
\Longrightarrow & (\exists T,\quad \textit{alloc}'. \\
 & \qquad \textit{consis}(\textit{alloc}', \textit{tenv}, \textit{pt}, c', \textit{asm}^{T}) \wedge \\
 & \qquad \textit{valid-c0}(\textit{tenv}, \textit{pt}, c') \wedge \\
 & \qquad \textit{isa-asm-precond}_{\textit{init}}(\textit{asm}^{T}, \textit{code-base}, \textit{code-size}(\textit{tenv}, \textit{pt}, c.\mathsf{mf}.\mathsf{gm})) \wedge \\
 & \qquad \textit{isa-asm-precond}^{p}_{\textit{dyn}}(\textit{asm}, \textit{code-base}, \textit{code-size}(\textit{tenv}, \textit{pt}, c.\mathsf{mf}.\mathsf{gm}), N) \wedge \\
 & \qquad \textit{asm}^{T}.\mathsf{spr} = \textit{asm}.\mathsf{spr})
\end{array}$$

This theorem has been formally proven by Leinenbach [Lei08].

Extension to Devices

First, we show that all variables are stored on the consistent target machine within the range specified by the *sufficient-memory* predicate:

Lemma 16 (Variable Allocation in Range)

$$\begin{array}{l}
\textit{consis}(\textit{alloc}, \textit{tenv}, \textit{pt}, c, d) \wedge \\
\textit{sufficient-memory}(\textit{tenv}, \textit{pt}, c, \textit{max-address}) \quad \Longrightarrow \\
\forall x \in \textit{valid-vars}(c).\ \textit{alloc}(x) < \textit{max-address}
\end{array}$$

Assuming that the heap memory does not overlap with the memory region which is mapped to device ports, the next Lemma claims that the compiled code never accesses devices.

Lemma 17 (Compiled Code does not Access Devices)

$$\begin{array}{ll}
 & \mathit{consis}(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c, \mathit{asm}) \\
\wedge & \mathit{max\text{-}address} < \mathsf{device\text{-}border} \\
\wedge & \mathit{valid\text{-}c0}(\mathit{tenv}, \mathit{pt}, c) \\
\wedge & \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asm}, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}, c.\mathsf{mf}.\mathsf{gm})) \\
\wedge & (\forall i < N.\ \mathit{sufficient\text{-}memory}(\mathit{tenv}, \mathit{pt}, \mathsf{the}(\delta^{i}_{C0}(c, \mathit{tenv}, \mathit{pt})), \mathit{max\text{-}address})) \\
\wedge & \delta^{N}_{C0}(c, \mathit{tenv}, \mathit{pt}) = \lfloor c' \rfloor \\
\Longrightarrow & (\exists T, \quad \mathit{alloc}'. \\
 & \qquad \mathit{consis}(\mathit{alloc}', \mathit{tenv}, \mathit{pt}, c', \mathit{asm}^{T}) \wedge \\
 & \qquad (\forall t < T. \mathit{da}(\mathit{asm}^{t}) = P))
\end{array}$$

The lemma is proven by induction on the program rest of c. The only relevant cases are accesses to variables which map to memory accesses on the target machine. By the definition of *sufficient-memory* we conclude that all memory frames are allocated within *max-address* and Lemma 16 ensures that all variable accesses do not overlap with device memory.

Now, we can extend compiler correctness to the VAMP assembly with devices. In short we want to ensure that for all possible execution sequences, devices do not interfere with the compiled code. Additionally to the conditions of the compiler correctness we only require that the space needed for executing the compiled machine does not overlap with the memory domain of devices. Note, that we also claim that the compiled code does not interfere with the computations of devices.

Theorem 5 (Extended Compiler Correctness)

$$\begin{array}{ll}
 & \mathit{consis}(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c, \mathit{asmd}.\mathsf{proc}) \\
\wedge & \mathit{max\text{-}address} < \mathsf{device\text{-}border} \\
\wedge & \mathit{valid\text{-}c0}(\mathit{tenv}, \mathit{pt}, c) \\
\wedge & \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asmd}.\mathsf{proc}, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}, c.\mathsf{mf}.\mathsf{gm})) \\
\wedge & (\forall i < N.\ \mathit{sufficient\text{-}memory}(\mathit{tenv}, \mathit{pt}, \mathsf{the}(\delta^{i}_{C0}(c, \mathit{tenv}, \mathit{pt})), \mathit{max\text{-}address})) \\
\wedge & \delta^{N}_{C0}(c, \mathit{tenv}, \mathit{pt}) = \lfloor c' \rfloor \\
\Longrightarrow & (\forall \quad \mathit{seq} \in \mathit{Seq}_{V}.\exists T, \mathit{alloc}'. \\
 & \quad \mathit{consis}(\mathit{alloc}', \mathit{tenv}, \mathit{pt}, c', \mathit{fst}(\mathit{asmd}^{\mathit{seq},T}).\mathsf{proc}) \wedge \\
 & \quad \mathit{valid\text{-}c0}(\mathit{tenv}, \mathit{pt}, c') \wedge \\
 & \quad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{fst}(\mathit{asmd}^{\mathit{seq},T}).\mathsf{proc}, \mathit{code\text{-}base}, \\
 & \qquad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}, c.\mathsf{mf}.\mathsf{gm})) \wedge \\
 & \quad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{dyn}}(\mathit{asmd}, \mathit{code\text{-}base}, \\
 & \qquad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}, c.\mathsf{mf}.\mathsf{gm}), N) \wedge \\
 & \quad \mathit{fst}(\mathit{asmd}^{\mathit{seq},T}).\mathsf{proc}.\mathsf{spr} = \mathit{asmd}.\mathsf{proc}.\mathsf{spr} \wedge \\
 & \quad (\forall \mathit{id}_X \in \mathfrak{D}\,. \\
 & \qquad \mathit{asmd}^{\mathit{seq},T}.\mathsf{devs}(\mathit{id}_X) = \mathit{asmd}^{\pi(\mathit{seq},\mathit{id}_X),\mathit{step\text{-}nr}(\mathit{seq},\mathit{id}_X,T)}.\mathsf{devs}(\mathit{id}_X)))
\end{array}$$

By Lemma 17 we can deduce processor locality of the compiled code. Then we apply Lemma 5 of the reordering theory to lift correctness from processor local computations to arbitrary ones. The last conjunct, claiming non-interference to device computations, is discharged by an application of Theorem 3.

3.3.8 Estimating Memory Consumption

Any code verification effort claiming the label *pervasive* has to deal with the concrete memory consumption of the given program and with memory restrictions of the target machine. C0 compiler correctness, for example, is only applicable if in each step sufficient heap and stack memory is granted for the assembly machine.

However, almost always, such assumptions are silently ignored because they are not visible in the semantics of the given high-level language. As in the C0 small-step semantics, those models assume some infinite memory. Memory restrictions emerge not until results are propagated down to lower-levels (as for example to assembly or the instruction set architecture).

Even though not required at the level of programming language semantics, reasoning about memory restrictions should utilize abstractions provided by the high-level C0 semantics, as done in Section 3.3.7.

In this section we develop a small theory on how to discharge assumptions on memory restrictions. We start with the simplest and most coarse estimation of the heap consumption, assuming the absence of garbage collection. Next, dealing with the stack memory, we propose two different strategies for estimating an upper bound of the consumption:

- Dynamically. By having an upper bound of the recursion depth, we can estimate the overall stack consumption of the machine. This upper bound can be determined by a simple extension of traditional programming logic. In the following we do not elaborate on this approach any further.
- Statically. An upper bound of the stack size of functions with no recursive calls can be determined by a statical analysis of the function body (as defined by A. Tsyban). In this section we present this approach and the corresponding soundness arguments.

Estimating Heap Consumption

If no garbage collection is running the heap consumption of a C0 program will steadily increase. The reason is simple: C0 offers only a construct to allocate new memory on the heap, but not to set memory free again.

Lemma 18 (Monotonicity of Heap Consumption in C0) *Given a non stuck computation $\delta^i_{C0}(c, tenv, pt) = \lfloor c' \rfloor$, then the following holds:*

$$\textit{heap-size}(c, tenv) \leq \textit{heap-size}(c', tenv)$$

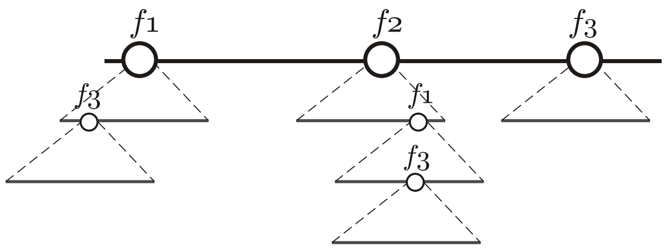


Figure 3.9: Invocation tree of a statement – *lines*: statements; *circles*: function calls

This lemma is proven by natural induction on i. The induction step follows by structural induction on the program rest.

Thus, in the compiler correctness theorem (without garbage collection) it suffices to claim the *sufficient-heap* predicate only for the final configuration.

Estimating stack consumption statically

In short, this approach determines the deepest (and most stack consuming) path in the invocation tree of a given function. In an invocation tree nodes are annotated with function names and the children of a node are given by the names of the top-level functions invoked in the parent's body (see Figure 3.9). This tree is finite, if none of the invoked functions is recursive (this includes also indirect recursion). The cost of a path in that tree is the sum of frame sizes of all functions appearing on that path.

Our goal is to prove that the cost of the most costly path is an upper bound for the stack consumption during the execution of the root function. Even though this claim seems to be reasonably intuitive, its proof turns out to be quite tricky.

Given a procedure table *pt* and a function name *fn* we inductively define the set of valid upper bounds for all costs of paths rooted in *fn* by:

$$\mathit{ub\text{-}costs} :: \mathit{tenv}_T \times \mathit{proctable}_T \times \mathit{fun}_n \to (\mathbb{N}\ \mathit{set})$$

$$\frac{\begin{array}{ll}(\forall \mathit{fn}' \in \mathit{top\text{-}scalls}(\mathit{pt}(\mathit{fn}).\mathsf{body}). & \\ \quad (\exists sz'.\ \ sz' \in \mathit{ub\text{-}costs}(\mathit{tenv}, \mathit{pt}, \mathit{fn}') \wedge & \mathit{pt}(\mathit{fn}) \neq \mathit{undef} \\ \qquad\quad sz' + \mathit{to\text{-}target}(\mathit{fun\text{-}size}(\mathit{tenv}, \mathit{pt}(\mathit{fn}))) \leq sz)) & \end{array}}{sz \in \mathit{ub\text{-}costs}(\mathit{tenv}, \mathit{pt}, \mathit{fn})}$$

Note, that this definition was originally introduced by A. Tsyban and subsequently adapted by the author. Example: Consider Figure 3.9. Let fs_1, fs_2 and fs_3 denote the frame size of the respective functions. Then we can infer

$(fs_1 + fs_3 + fs_2) \in \textit{ub-costs}(tenv, pt, f_2)$ as follows:

$$\dfrac{\dfrac{\dfrac{\textit{top-scalls}(pt, f_3) = \{\}}{fs_3 \in \textit{ub-costs}(tenv, pt, f_3)} \quad \textit{top-scalls}(pt, f_1) = \{f_3\}}{(fs_1 + fs_3) \in \textit{ub-costs}(tenv, pt, f_1)} \quad \textit{top-scalls}(pt, f_2) = \{f_1, f_3\}}{(fs_2 + fs_1 + fs_3) \in \textit{ub-costs}(tenv, pt, f_2)}$$

Note that the definition is sensitive to recursion, i.e. if either fn or one of the nested functions appearing in the code of fn is recursive, then the corresponding set of upper bounds is empty: $\textit{ub-stack}(tenv, pt, fn) = \{\}$.

The predicate *valid-ub-costs* yields whether a given upper bound holds for a set of functions:

$$\textit{valid-ub-costs} :: tenv_T \times proctable_T \times stmt \times \mathbb{N} \rightarrow bool$$

$$\textit{valid-ub-costs}(tenv, pt, F, ub) \equiv \forall fn \in F.\ ub \in \textit{ub-costs}(tenv, pt, fn)$$

Upper bounds on costs can also be defined directly on some statement p by:

$$\textit{valid-ub-costs}(tenv, pt, \textit{top-scalls}(p), ub)$$

Now we want to relate upper bounds of costs to the stack memory consumption during the execution:

Theorem 6 (Upper bound for stack memory consumption) *Given a non stuck computation* $\delta^i_{C0}(c, tenv, pt) = \lfloor c' \rfloor$*, then the following holds:*

$$\begin{array}{ll} & c.\mathsf{prog} = \mathsf{SCall}(vn, fn, pl) \\ \wedge & (MAX - \textit{local-sz}(c.\mathsf{mf}.\mathsf{lms})) \in \textit{ub-costs}(tenv, pt, fn) \\ \Longrightarrow & \textit{local-sz}(c'.\mathsf{mf}.\mathsf{lms}) \leq MAX \end{array}$$

At a first glance, the prove of this theorem seems to be straightforward: we only have to find a suitable inductive invariant, i.e. a predicate which implies the conclusion and is invariant under C0 transitions. As a first guess, we choose as invariant:

$$\textit{valid-ub-costs}(tenv, pt, \textit{top-scalls}(c.\mathsf{prog}), (MAX - \textit{local-sz}(c.\mathsf{mf}.\mathsf{lms})))$$

Unfortunately, this predicate is not inductive. Consider again the example in Figure 3.9, where the first line should be the function body of fn: Suppose that the next statement to execute in the program rest is the first invocation of f_1. Furthermore, suppose the predicate *valid-ub-costs* holds for the configuration directly after the invocation of f_1 the function call is substituted by the function body and a new frame is allocated in the stack. To maintain the invariant, the upper bound for the costs must decrease because the stack size increased. This, however, is not true, because *valid-ub-costs* is determined by the invocation of the function with the highest cost, i.e. by f_2, which is still present in the new program rest.

Thus, the predicate is too weak: For those functions invoked within f_1 we need a different invariant than for those invoked after it returns. For functions called after the return statement there will be more memory available and therefore the upper bound may increase again. The solution is to define a sequence of invariants – one for each not yet returned function body in the current program rest.

The boundaries of these invariants are return statements: function names appearing between two consecutive return statements are grouped together. This is done by the function *return-fn*, which returns for a given statement a list of sets of function names. The first set of this list contains all function names appearing in the statement before the first return is encountered, the second set all those function names appearing between the first and the second return, and so on.

To define *return-fn*, we need a helper function, with an accumulator argument:

$$\textit{return-fns} :: \textit{stmt list} \times (\textit{fun}_n\ \textit{set}) \rightarrow (\textit{fun}_n\ \textit{set})\ \textit{list} \times (\textit{fun}_n\ \textit{set})$$

```
$return\text{-}fns([], s)$       $=$  $([], s)$
$return\text{-}fns((h\#t), s)$  $=$
  case $h$ of
    $(\mathsf{Return}(e))$            $\Longrightarrow$ (  let
                                            $(l', s') = return\text{-}fns(t, \{\})$
                                        in
                                            $(s\#l', s'))$
    $(\mathsf{SCall}(vn, fn, pl))$  $\Longrightarrow$     $(return\text{-}fns(t, (s \cup \{fn\})))$
    $(else)$                          $\Longrightarrow$     $return\text{-}fns(t, (s \cup top\text{-}scalls(h)))$
```

Now *return-fn* is defined as follows:

$$\textit{return-fn} :: \textit{stmt} \rightarrow (\textit{fun}_n\ \textit{set})\ \textit{list}$$

```
$return\text{-}fn(p) =$  let
                            $(l', s') = return\text{-}fns(s2l(p), \{\})$
                        in
                            $l'@[s']$
```

The sequence of invariants described above is now easy to state: given a local memory frame *lms*, a list of sets of function names, and an upper bound *MAX* for local memory consumption we require that costs of the functions of the ith set never exceed the memory currently available. In each recursion step the currently available memory increases, by dropping the head stack frame of *lms*.

$$\textit{stack-inv} :: \textit{tenv}_T \times \textit{proctable}_T \times \mathbb{N} \times \textit{mem-frame}_T\ \textit{list} \times (\textit{fun}_n\ \textit{set})\ \textit{list} \rightarrow \textit{bool}$$

$$\begin{aligned}
&\mathit{stack\text{-}inv}(\mathit{tenv}, \mathit{pt}, \mathit{MAX}, \mathit{lms}, []) = \mathit{MAX} \geq \mathit{local\text{-}sz}(\mathit{lms})\\
&\mathit{stack\text{-}inv}(\mathit{tenv}, \mathit{pt}, \mathit{MAX}, \mathit{lms}, h\#t) =\\
&\qquad \mathit{stack\text{-}inv}(\mathit{tenv}, \mathit{pt}, \mathit{MAX} - \mathit{to\text{-}target}(\mathit{frame\text{-}size}(\mathit{tenv}, \mathit{hd}(\mathit{lms}))), \mathit{tl}(\mathit{lms}), t)\\
&\quad\wedge \quad \mathit{valid\text{-}ub\text{-}costs}(\mathit{tenv}, \mathit{pt}, h, \mathit{MAX} - \mathit{local\text{-}sz}(\mathit{lms}))
\end{aligned}$$

The predicate *stack-inv* is invariant under C0 transitions:

Lemma 19 (Invariance of Stack-inv) *Given a non stuck computation $\delta_{c0}(c, \mathit{tenv}, \mathit{pt}) = \lfloor c' \rfloor$, then the following holds:*

$$\begin{aligned}
&\mathit{stack\text{-}inv}(\mathit{tenv}, \mathit{pt}, \mathit{MAX}, \mathit{lms}, \mathit{return\text{-}fn}(c.\mathsf{prog}))\\
\Longrightarrow\ &\mathit{stack\text{-}inv}(\mathit{tenv}, \mathit{pt}, \mathit{MAX}, \mathit{lms}, \mathit{return\text{-}fn}(c'.\mathsf{prog}))
\end{aligned}$$

This lemma is proven by structural induction on the program rest of c. The correctness arguments for the different statements are:

- $\mathsf{SCall}(\mathit{vn},\mathit{fn},\mathit{pl})$: The first element in the invariant sequence requires the costs of fn to be bounded by: $(\mathit{MAX} - \mathit{local\text{-}sz}(\mathit{lms})) \in \mathit{ub\text{-}costs}(\mathit{tenv}, \mathit{pt}, \mathit{fn})$. After the transition a new element lm is added to the stack and thus the new invariant for the function body must hold:

  $$\mathit{valid\text{-}ub\text{-}costs}(\mathit{tenv}, \mathit{pt}, \mathit{top\text{-}scalls}(\mathit{pt}(\mathit{fn}).\mathsf{body}), \mathit{MAX} - \mathit{local\text{-}sz}(\mathit{lm}@\mathit{lms}))$$

  By the definition of *ub-costs* the upper bound for functions in the function body decreases by the size of the function declaration. This size coincides, by Lemma 15, with the size of the newly allocated stack frame lm.

- $\mathsf{Return}(\mathit{exp})$: The top most stack frame is deleted and new stack memory is available. At the same time, the top most invariant is dropped. Hence, we have to show:

  $$\mathit{stack\text{-}inv}(\mathit{tenv}, \mathit{pt}, \mathit{MAX}, \mathit{tl}(\mathit{lms}), \mathit{tl}(\mathit{return\text{-}fn}(c.\mathsf{prog})))$$

  This follows by unpacking the recursive assumption *stack-inv* once.

- $\mathsf{Comp}(s_1, s_2)$: Proved by applying the induction hypothesis on s_1. The proof is highly technical and not detailed here.

- *Others*: Neither the stack size nor the list of sets $\mathit{return\text{-}fn}(c'.\mathsf{prog})$ changed.

From this lemma and the following two Lemmas 20 and 21, Theorem 6 can be easily deduced.

The invariant can be established by the assumption of the theorem:

Lemma 20 (Establishing the stack invariant)

$$\begin{aligned}
&c.\mathsf{prog} = \mathsf{SCall}(\mathit{vn}, \mathit{fn}, \mathit{pl})\\
\wedge\ &(\mathit{MAX} - \mathit{stack\text{-}size}(\mathit{tenv}, c)) \in \mathit{ub\text{-}costs}(\mathit{tenv}, \mathit{pt}, \mathit{fn})\\
\Longrightarrow\ &\mathit{stack\text{-}inv}(\mathit{pt}, \mathit{MAX}, \mathit{lms}, \mathit{return\text{-}fn}(c.\mathsf{prog}))
\end{aligned}$$

This lemma is proven by unpacking the definition of *stack-inv*.

The conclusion of the theorem follows from the invariant:

Lemma 21 (Relation of *stack-inv* and *local-sz*)

$$\begin{array}{rl} & \mathit{stack\text{-}inv}(pt, MAX, \mathit{lms}, \mathit{return\text{-}fn}(c.\mathsf{prog})) \\ \Longrightarrow & \mathit{local\text{-}sz}(\mathit{lms}) \leq MAX \end{array}$$

This lemma is proven by induction on the size of the list $\mathit{return\text{-}fn}(c.\mathsf{prog})$.

Note, that the author of this thesis thanks Dirk Leinenbach, who contributed substantially to formalize the proof of Theorem 6 in Isabelle/HOL.

3.3.9 Some Remarks

The value representation in C0 is not very close to what you would expect from the semantics of a high-level programming language. Indeed, talking of values at all in the context of C0 small-step semantics is misleading: variables are stored in (word-addressable) memories. Hence, even a simple equality check between two evaluated expressions becomes an unnecessarily complicated task (one needs the type of the value to deduce the number of relevant memory cells in the provided content function).

For compiler verification this may be a reasonable decision, since data representation is kept quite close to the one given on the target machine. However, verifying programs directly on the semantics turns out to be hardly manageable.

The proponents of the semantics will plead, that it was never designed for code verification, as for this purpose a Hoare logic on a more abstract representation of C0 is provided. And would not a translation of properties to the concrete small-step semantics be possible by a meta theorem which is proven sound once and for all? Unfortunately, the answer is no. One reason is that the abstract representation is shallow embedded into Isabelle, i.e. expressions of the programming languages are simply Isabelle expressions, where the concrete C0 small-step semantics is deep embedded, i.e. expressions are defined by some abstract data type. Hence, any automatic property transfer would result in automatic generation of lemmas.

Yet, there is another, more subtle reason, why the proposed scheme — an explicit small-step model for compiler verification, and an abstract logic for program verification — turned out to be cumbersome in practice. When proving the correctness of system code, especially if the code covers inline assembly portions (as described in the next section), we often have to switch between different semantical layers — even during one function call. Thus, code verification of all C0 parts can not be carried out in one step, rather many (tedious) property transfers are necessary.

3.4 C0 With Inline Assembly

3.4.1 Syntax

We embed inline assembly by the new statement, denoted with Asm. This statement takes a list of assembly instructions.

$$stmt ::= \ldots \mid \mathsf{Asm}\ \mathbf{of}\ instr_T\ list$$

3.4.2 Configuration

Of course, the subject of C0 with inline assembly semantics can not be solely C0. Rather, we define a combined configuration of type:

$$C_{c+a} = C_{\mathrm{co}} \times C_{asm}$$

3.4.3 Semantics

Reasoning about the execution of a C0 with inline assembly computation starting at configuration (c, asm) is straightforward: Ordinary C0 statements are executed according to the small-step semantics defined in 3.3.5. When an assembly statement $\mathsf{Asm}(il)$ is met we switch to a consistent assembly state asm with the dpc pointing to the program il. This switching is justified by compiler correctness and Lemma 22. Next we process program il according to the VAMP assembly semantics. The newly obtained assembly state asm' is finally matched again to a consistent C0 configuration c'. Note, that this is not always possible: e.g. the assembly program must not change certain registers (e.g. the stack pointer). In Lemma 23 we show how to construct a consistent C0 configuration out of the new and the old assembly and the old C0 configurations.

The following lemma states the correspondence between the instruction list of an assembly statement in the C0 program and in the code, pointed to by the program counter of a consistent assembly machine:

Lemma 22 (Control Consistency for Inline Assembly)

$$\begin{array}{rl} & consis(alloc, tenv, pt, c, asm) \\ \wedge & hd(s2l(c.\mathsf{prog})) = \mathsf{Asm}(il) \\ \Longrightarrow & to\text{-}instr\text{-}list(asm.\mathsf{mm}, asm.\mathsf{dpc}, length(il)) = il \end{array}$$

If we restrict the assembly code only to alter variables of elementary type, and neither to modify the code nor certain control registers, then the function *c0-asm-update* describes the effect of that code in terms of a C0 state update.

$$\begin{array}{c} c0\text{-}asm\text{-}update :: \\ tenv_T \times C_{\mathrm{co}} \times C_{asm} \times C_{asm} \times (var\text{-}id)\ list \rightarrow C_{\mathrm{co}}\ option \end{array}$$

The function takes as input the type environment of the C0 machine, the original C0 and a consistent assembly configuration, the new assembly configuration and a list gl of identifiers of variables that has changed. If one of the following conditions is not fulfilled then the function returns with $c0\text{-}asm\text{-}update(tenv, c, asm, asm', gl) = \bot$.

- The new assembly machine finished the execution of the instruction list, i.e. $asm'.\mathsf{dpc} = asm.\mathsf{dpc} + length(il) \;\wedge\; asm'.\mathsf{pc} = asm'.\mathsf{dpc} + 4$.
- Code region of the C0 program has not been modified:

$$\begin{array}{rl} & to\text{-}instr\text{-}list(asm'.\mathsf{mm}, code\text{-}base, cs) \\ = & to\text{-}instr\text{-}list(asm.\mathsf{mm}, code\text{-}base, cs) \end{array}$$

  Where cs denotes the code size in bytes, i.e. $cs = code\text{-}size(tenv, pt, c.\mathsf{gm})$.
- The registers pointing to the stack start, to the top most stack frame and the next heap element to allocate have not changed:

$$\begin{array}{lcl} asm.\mathsf{gpr}[\mathsf{sbase}] & = & asm'.\mathsf{gpr}[\mathsf{sbase}] \\ asm.\mathsf{gpr}[\mathsf{toplm}] & = & asm'.\mathsf{gpr}[\mathsf{toplm}] \\ asm.\mathsf{gpr}[\mathsf{toph}] & = & asm'.\mathsf{gpr}[\mathsf{toph}] \end{array}$$

- All variables in gl are valid and elementary:

$$\forall x \in set(gl)\ .\ x \in valid\text{-}vars(c) \;\wedge\; is\text{-}elementary(x)$$

- Out of the stack and heap memory of the C0 machine, the assembly code only manipulated addresses mapping to variables in gl:

$$\begin{array}{l} \forall ad \leq heap\text{-}base + heap\text{-}size(tenv, c). \\ \qquad stack\text{-}start(tenv) \leq ad \implies \\ (asm.\mathsf{mm}(ad) \neq asm'.\mathsf{mm}(ad) \\ \qquad\qquad \implies \exists x \in set(gl).alloc(x) = ad) \end{array}$$

Note that during an assembly computation these conditions do not have to hold, e.g. all registers may be written. The programmer only has to ensure that with the execution of the last instruction, the conditions for construction are fulfilled.

If the conditions are all satisfied the function returns a new C0 configuration $\lfloor c' \rfloor = c0\text{-}asm\text{-}update(tenv, c, asm, asm', gl)$, which differs from the original one by

- the program rest: The assembly statement is substituted by a Skip, i.e.

$$s2l(c'.\mathsf{prog}) = [\mathsf{Skip}]@tl(s2l(c.\mathsf{prog}))$$

- the memory configuration: all variables in the list gl are updated by the values read out of the new assembly configuration.

$$\mathsf{the}(\mathit{lookup}(c'.\mathsf{mf}, gl[i])).\mathsf{content}(0) = \mathit{asm}'.\mathsf{mm}(\mathit{alloc}(gl[i]))$$

The next lemma links the result of the assembly computation and the constructed C0 configuration. Note, that compiler consistency can not always be established directly after the instruction list is executed. For example if the assembly statement was the last statement in a loop, the program counters have first to be reset. Therefore, we introduce a weaker form of the predicate *consis*, denoting that two states are *almost* compiler consistent:

$$\begin{aligned} &\mathit{consis}'(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c, \mathit{asm}) \equiv \\ &\quad (\exists t.\quad \mathit{consis}(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c, \mathit{asm}^t) \wedge \\ &\qquad \mathit{asm}.\mathsf{gpr} = \mathit{asm}^t.\mathsf{gpr} \wedge \\ &\qquad \mathit{asm}.\mathsf{spr} = \mathit{asm}^t.\mathsf{spr} \wedge \\ &\qquad \mathit{asm}.\mathsf{mm} = \mathit{asm}^t.\mathsf{mm}) \end{aligned}$$

Then, the described lemma reads as follows:

Lemma 23 (Consistent Update)

$$\begin{array}{ll} & \mathit{consis}(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c, \mathit{asm}) \\ \wedge & \mathit{valid\text{-}c0}(\mathit{tenv}, \mathit{pt}, c) \\ \wedge & \mathit{c0\text{-}asm\text{-}update}(\mathit{tenv}, c, d, d', gl) = \lfloor c' \rfloor \\ \Longrightarrow & \mathit{consis}'(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c', \mathit{asm}') \wedge \\ & \mathit{valid\text{-}c0}(\mathit{tenv}, \mathit{pt}, c') \end{array}$$

The formal proof of this lemma was conducted by Tsyban in [Tsy09].

Utilizing the lemma above, we can define a transition $\rightarrow_{\mathrm{c+a}}$ relation for C0 with inline assembly code (cf. Figure 3.4.3). Two rules are required, one for ordinary C0 statements and one for inline assembly portions.

The next lemma, states that the transition relation is sound, i.e. it complies with plain assembly semantics. With $\rightarrow^*_{\mathrm{c+a}}$ we denote the reflexive, transitive closure of $\rightarrow_{\mathrm{c+a}}$.

Theorem 7 (Soundness of inline assembly transitions)

$$\begin{array}{ll} & \mathit{consis}(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c, \mathit{asm}) \\ \wedge & \mathit{valid\text{-}c0}(\mathit{tenv}, \mathit{pt}, c) \\ \wedge & \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asm}, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}, c.\mathsf{mf}.\mathsf{gm})) \\ \wedge & (c, \mathit{asm}) \rightarrow^*_{c+a} (c', \mathit{asm}') \\ \Longrightarrow & (\exists T\,.\, \mathit{consis}'(\mathit{alloc}, \mathit{tenv}, \mathit{pt}, c', \mathit{asm}^T) \wedge \\ & \mathit{valid\text{-}c0}(\mathit{tenv}, \mathit{pt}, c') \wedge \\ & \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asm}^T, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}, c.\mathsf{mf}.\mathsf{gm})) \wedge \\ & \mathit{isa\text{-}asm\text{-}precond}^p_{\mathit{dyn}}(\mathit{asm}, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}, c.\mathsf{mf}.\mathsf{gm}), T)) \end{array}$$

$$\frac{\begin{array}{c} hd(s2l(c.\mathsf{prog})) \neq \mathsf{Asm}(\dots) \\ \textit{sufficient-memory}(tenv, pt, c, \textit{max-address}) \\ consis(alloc, tenv, pt, c', asm') \end{array} \qquad \delta_{c0}(c, tenv, pt) = \lfloor c' \rfloor}{(c, asm) \rightarrow_{\text{c+a}} (c', asm')}$$

$$\frac{\begin{array}{c} hd(s2l(c.\mathsf{prog})) = \mathsf{Asm}(il) \\ \exists gl\,.\,\lfloor c' \rfloor = \textit{c0-asm-update}(tenv, c, asm, asm', gl) \end{array} \qquad \exists t\,.\,asm' = asm^t}{(c, asm) \rightarrow_{\text{c+a}} (c', asm')}$$

Figure 3.10: Transition Relation $\rightarrow_{\text{c+a}}$ for C0 with inline assembly computations

The theorem is proven by applying the verification scheme for inline assembly code introduced at the beginning of this section, i.e. by using Lemma 23 and Lemma 22.

Note, that completeness is not shown, i.e. not all assembly computations can be expressed in our semantics. For example, inline code which manipulates the stack pointer is not covered.

3.4.4 Extension to Devices

In this section, we introduce a semantics to ease the verification of C0 programs with inline assembly drivers, where each assembly portion is controlling at most one device. We prove soundness of the semantics by applying the reordering theorem proven in Section 3.2.2.

The transition system $\rightarrow_{\text{c+ad}}$ is defined in Figure 3.4.4. We distinguish three cases: (i) the next C0 statement is ordinary, i.e. not an inline assembly portion; then the device states are not altered, (ii) the next statement is an inline assembly portion, which does not access devices; then again device states stay unchanged, (iii) the next statement is an inline assembly portion, which only accesses a certain device id_X, i.e. which is *pure*; then it suffices to verify the assembly portion for execution sequences, that are reduced to processor steps and steps of device id_X. We denote such sequences by $Seq_V(id_X) \equiv \{seq \mid seq' \in Seq_V \wedge seq = \mathit{filter}(seq, (\lambda ev\,.\,ev = P \vee ev = id_X))\}$. Note, that then all devices except for id_X are not modified.

The big advantage of this semantics is that it enables us to separate verification of C0 and inline assembly parts, and of assembly drivers for different devices. It implicitly states, that compiled C0 code does not interfere with devices and that a driver for one device does not interfere with steps of other devices, since device steps are always reordered to the corresponding driver.

For propagating properties proven in the above defined transition system down to the level of VAMP assembly with devices, we again need to prove

$$\frac{\begin{array}{cc} hd(s2l(c.\mathsf{prog})) \neq \mathsf{Asm}(\dots) & \delta_{c0}(c, tenv, pt) = \lfloor c' \rfloor \\ \mathit{sufficient\text{-}memory}(tenv, pt, c, max\text{-}address) & asmd'.\mathsf{devs} = \\ consis(alloc, tenv, pt, c', asmd'.\mathsf{proc}) & asmd.\mathsf{devs} \end{array}}{(c, asmd) \rightarrow_{\text{c+ad}} (c', asmd')}$$

$$\frac{\begin{array}{c} hd(s2l(c.\mathsf{prog})) = \mathsf{Asm}(il) \\ \exists gl\,.\, \lfloor c' \rfloor = c0\text{-}asm\text{-}update(tenv, c, asmd.\mathsf{proc}, asmd'.\mathsf{proc}, gl) \\ \exists T\,.\, asmd'.\mathsf{proc} = (asmd.\mathsf{proc})^T \wedge \forall t < T\,.\, da((asmd.\mathsf{proc})^t) = P \\ asmd'.\mathsf{devs} = asmd.\mathsf{devs} \end{array}}{(c, asmd) \rightarrow_{\text{c+ad}} (c', asmd')}$$

$$\frac{\begin{array}{c} hd(s2l(c.\mathsf{prog})) = \mathsf{Asm}(il) \\ \exists gl\,.\, \lfloor c' \rfloor = c0\text{-}asm\text{-}update(tenv, c, asmd.\mathsf{proc}, asmd'.\mathsf{proc}, gl) \\ \forall seq \in Seq_V(id_X)\,.\, \exists T\,.\, pure(asmd, seq, T, id_X) \wedge asmd' = asmd^{seq,T} \end{array}}{(c, asmd) \rightarrow_{\text{c+ad}} (c', asmd')}$$

Figure 3.11: Transition Relation $\rightarrow_{\text{c+ad}}$ for C0 with inline assembly drivers

a soundness theorem. We show that a computation in the transition system $\rightarrow^*_{c+ad}$ has for any valid execution sequence a counter part in the VAMP assembly with devices model. Note, however, that the latter computation will not necessary lead to the same device states as the former one. That is, because in $\rightarrow_{c+ad}$ devices only make transitions during assembly portions. However, assuming that the final device state is *stable*, we can establish equality:

Theorem 8 (Soundness of transition system for inline assembly driver)

$$\begin{array}{ll} & consis(alloc, tenv, pt, c, asmd.\mathsf{proc}) \\ \wedge & valid\text{-}c0(tenv, pt, c) \\ \wedge & isa\text{-}asm\text{-}precond_{init}(asmd.\mathsf{proc}, code\text{-}base, code\text{-}size(tenv, pt, c.\mathsf{mf.gm})) \\ \wedge & (c, asmd) \rightarrow^*_{c+ad} (c', asmd') \\ \Longrightarrow & (\forall seq \in Seq_V\,.\, \exists T\,. \\ & \quad asmd'.\mathsf{proc} = asmd^{seq,T}.\mathsf{proc} \wedge \\ & \quad consis(alloc, tenv, pt, c', asmd'.\mathsf{proc}) \wedge \\ & \quad valid\text{-}c0(tenv, pt, c') \wedge \\ & \quad isa\text{-}asm\text{-}precond_{init}(asmd^{seq,T}, code\text{-}base, \\ & \quad\quad code\text{-}size(tenv, pt, c.\mathsf{mf.gm})) \wedge \\ & \quad isa\text{-}asm\text{-}precond_{dyn}(asmd, code\text{-}base, \\ & \quad\quad code\text{-}size(tenv, pt, c.\mathsf{mf.gm}), seq, T)) \wedge \\ & \quad (\forall id_X \in \mathfrak{D}\,. \\ & \quad\quad stable(asmd'.\mathsf{devs}(id_X)) \Longrightarrow \\ & \quad\quad asmd'.\mathsf{devs}(id_X) = asmd^{seq,T}.\mathsf{devs}(id_X)) \end{array}$$

The theorem is proven by induction on the step numbers of the computation $\rightarrow^*_{c+ad}$. The induction base is trivially true. For the induction step, we have a case-split on the three transition rules:

- For ordinary C0 statements, correctness follows from applying extended compiler correctness, i.e. Theorem 5.
- For assembly statements with no access devices, correctness follows from the application of Lemma 5. It ensures, that device states are not modified and that pure sequential assembly semantics can be applied.
- For assembly statements with device access, we apply the reordering theorem. Given a VAMP assembly computation in which the assembly statement is executed, then we reorder all device steps, which are not accessed, to the end of the computation.

Theorem 8 is only valid for devices which are finally in a stable state. This is a rather tough restriction. We can relax this condition by stating that not only stable states are preserved by both computations but also certain invariants on the final state.

We call a property $\Im$ over device states *external invariant* if the property is preserved under arbitrary external input:

$$Inv_{\text{ext}} = \{\Im \,|\, \forall eifi\,.\, \Im(c) \wedge (c', \dots) = \delta_{dsty(c)}(c, mifi_\epsilon, eifi) \implies \Im(c')\}$$

Thus, we can strengthen the soundness theorem by substituting the last conclusion in the theorem with:

$$\begin{aligned}\dots\wedge\quad & (\forall id_X \in \mathfrak{D}\,,\, \Im \in Inv_{\text{ext}}\,. \\ & \quad \Im(asmd'.\mathsf{devs}(id_X)) \implies \Im(asmd^{seq,T}.\mathsf{devs}(id_X)))\end{aligned}$$

The presented semantics is helpful when reasoning about C0 program correctness at the level of small-step semantics. However, combining these results with Hoare logic proofs is not straightforward. One obvious solution is to enrich the C0 Hoare logic with new rules dealing with inline assembly code and devices. These rules would look similar to the semantics presented here, and the corresponding soundness proof would follow the lines of the proof of Theorem 8. However, adding new rules to our logic involves non-trivial proof effort, since each new rule has to be justified on all different layers of the C0 language stack: Hoare logic, big step and small step.

We proceed in a different way: we encode the effects of assembly drivers as atomic state updates, called XCalls. These XCalls are introduced to all the models in the C0 language stack, up to the Hoare logic. There, an already defined general assignment rule is used to encode these atomic XCalls. Note, that the last two theorems haven't been formalized in Isabelle/HOL. Rather, in the formal work we applied the correctness arguments of the theorems manually.

3.4.5 Some Remarks

In a first approach, inline assembly was semantically embedded by defining a transition system on the combined state. The result of the execution of each inline assembly instruction had to be mapped back to a consistent C0 computation: most importantly store word instructions were mapped to corresponding C0 variable updates.

On the positive side, that approach provides a well defined small-step transition system, which always maintains full information on both machines. Yet, in most cases the state of a C0 machine during executing inline assembly is not of any advantage. Furthermore the additional information is bought by a loss of expressiveness: much rigor conditions on the execution of the inline assembly are required, e.g. storing half words is not allowed. In contrast, the new approach does not require any conditions on the execution of inline code at all. Only the result of the computation is of interest.

3.5 C0 With XCalls

The concept of *XCalls* is introduced to capture the semantical effects of function calls by atomic specifications. Particularly, when specifying functions with inline assembly portions, as for example drivers, the use of XCalls is appealing. First, we extend the C0 configuration by additional *meta variables*, representing those parts of the processor which are accessed by the assembly code. More general, these ghost variables — in the following called *extended state* — may abstract from arbitrary low-level entities which lie outside the scope of C0, e.g. memory, registers or even device states. An XCall describes the effect of a function call on the C0 configuration and on the extended state by one atomic state update. Compiler correctness remains applicable only in case implementation correctness proofs are provided for each of the XCalls.

The main charm of XCalls is that they enable us to argue on effects of inline assembly portions without caring about assembly semantics. Thus, by enriching the language stack with XCalls, we can lift assembly code and driver semantics up to the Hoare logic level. Then, by proving implementation correctness of XCalls we transfer results proven in Hoare logic down to VAMP assembly with devices. Note, that for drivers, XCalls abstract interleaved executions to sequential (and atomic) specifications. This is justified by the reordering theory developed previously.

In the following we first show how to extend configurations, statements, and the small-step semantics to integrate XCalls and then prove an extended compiler correctness theorem under the assumption that all XCalls are verified against their implementation. Finally, we sketch how traditional Hoare logic is extended to deal with XCalls.

3.5.1 Syntax

Syntactically XCalls are treated similar to function calls, with the main difference that they are not restricted to a single return value. The corresponding statement consists of (i) the XCall name, (ii) a list of names of variables to which the return values of the call are written, and (iii) a list of expressions denoting the parameters to be passed to the XCall:

$$\mathit{stmt} ::= \ldots \mid \mathsf{XCall}\ \mathbf{of}\ \mathit{fun}_n \times \mathit{var}_n\ \mathit{list} \times \mathit{expr}\ \mathit{list}$$

3.5.2 Configuration

The new configuration is parametrized over the type α of the extended component. Note, that with choosing α as an arbitrary Isabelle type, we do not restrict ourselves to C0 types. The record type $\alpha\ C_{\mathrm{cx}}$ consists of the two fields:

$\mathsf{c} \in C_{\mathrm{co}}$	The C0 configuration.
$\mathsf{xstate} \in \alpha$	The extended state.

In the following we refer to elements of type C_{cx} as extended configurations.

3.5.3 Semantics

The semantics of each XCall is described by a function of type

$$\alpha\ \mathit{xsem}_T = \mathit{content}_T\ \mathit{list} \times \alpha \to (\mathit{content}_T\ \mathit{list} \times \alpha)\ \mathit{option}$$

As input it takes a list of C0 contents and an extended state x and returns an updated extended state and a list of C0 contents to be assigned to the left expressions of the call. Note, that we allow the semantics to get stuck, i.e. to return with $\bot$.

Similar to a function declaration we define a *XCall declaration* as a triple of type

$$\alpha\ \mathit{xdecl}_T = \mathit{ty}\ \mathit{list} \times \mathit{ty}\ \mathit{list} \times \alpha\ \mathit{xsem}_T$$

The first two components denote the type of the parameters and of the return values respectively, where the last one stands for the call semantics. Analogous to a procedure table, we call a mapping from function names to XCall declarations a *XCall table*. It has type

$$\alpha\ \mathit{xtable}_T = \mathit{fun}_n \to \alpha\ \mathit{xdecl}_T$$

The transition function, takes as input the description of the extended machine, consisting of an extended configuration, a type environment, an ordinary procedure table and an XCall table. It returns either $\bot$ denoting a stuck computation, or the successor state of the extended C0 machine.

$$\delta_{cx} :: \alpha\ C_{\mathrm{cx}} \times \mathit{tenv} \times \mathit{proctable} \times \alpha\ \mathit{xtable}_T \to (\alpha\ C_{\mathrm{cx}})\ \mathit{option}$$

We define the transition function via the following case split:

- The statement to be executed is not an XCall. We apply the C0 transition function on the C0 state. If it returns with $\perp$ the whole transition gets stuck, otherwise the extended state remains unchanged und we get:

$$\delta_{cx}(c_X, tenv, pt, xpt) = (\mathsf{the}(\delta_{c0}(c_X.\mathsf{c}, tenv, pt)),\ c_X.\mathsf{xstate})$$

- The statement is an XCall, i.e. $c_X.\mathsf{c}.\mathsf{prog} = \mathsf{XCall}(fn, lvars, exps_p)$. First, we compute the following values:

$vas_p[i] = eval(c_X.\mathsf{c}, exps_p!i)$	The evaluation of each parameter expression for $0 \leq i < length(exps_p)$.
$(ty_p, ty_r, xsem) = xpt(fn)$	The XCall table contains a declaration for the invoked XCall, i.e. it does not return *undef*.
$(vals', c_X') = \mathsf{the}(xsem(vas_p, c_X.\mathsf{xstate}))$	We apply the specification of the XCall to the evaluated parameters and the extended configuration. As output we get a list of return values and the successor state.
$mf' = (upds_{\mathrm{mem}}(c_X.\mathsf{c}.\mathsf{mf}, lvars, vals')).\mathsf{mf}$	Finally, we assign the result values to the corresponding variables of the XCall. The function $upds_{\mathrm{mem}}(c, xs, vs)$ is defined recursively: it returns c if $xs = []$ and $upds_{\mathrm{mem}}(upd_{\mathrm{mem}}(c, x, v), xs', vs')$ for $xs = x\#xs'$ and $vs = v\#vs'$.

If any of the above expression evaluation, declaration mapping or application of semantics yields $\perp$ or is undefined, then the whole transition gets stuck. Otherwise, the result of the computation is:

$$\delta_{cx}(c_X, tenv, pt, xpt) = \lfloor((\mathsf{prog} = \mathsf{Skip}, \mathsf{mf} = mf'), c_X')\rfloor$$

We define the execution of n steps of the extended machine by:

$$\delta_{cx}{}^{n}(c_X, tenv, pt, xpt) = \begin{cases} \lfloor c_X \rfloor & \textbf{if } n = 0 \\ \delta_{cx}{}^{n-1}(c_X', tenv, pt, xpt) & \textbf{if } n \neq 0 \wedge \delta_{cx}(c_X, tenv, pt, xpt) = \lfloor c_X' \rfloor \\ \perp & \textbf{otherwise} \end{cases}$$

3.5.4 Correctness

In contrast to ordinary C0 programs, programs with XCalls cannot be compiled straight to assembly. Accordingly, properties proven for an extended machine are not linked solely by compiler correctness to corresponding properties on assembly. Rather we have to provide for each XCall (i) an implementation in C0 with inline assembly, (ii) a simulation relation, which maps the extended configuration to the implementation, and (iii) the corresponding correctness proof of the implementation.

Then, corresponding to the compiler correctness theorem, we can prove a simulation theorem. Basically, this theorem relates the execution of the extended C0 machine with the execution of the concurrent VAMP assembly with devices model. Furthermore, we maintain on the implementation side an intermediate C0 configuration which has to be consistent with the assembly state. We use an intermediate C0 machine because the extended C0 machine does not cover the whole C0 implementation and hence can not be related directly to the underlying assembly state (e.g. the C0 functions implementing the XCalls are not contained in the procedure table of the extended machine). Basically, the intermediate machine links the extended machine with the implementation of the XCalls.

Simulation Relation

Before stating the theorem, we define the corresponding abstraction relation. In short, it relates code and configurations of the extended C0 machine with the intermediate C0 machine and VAMP assembly with devices.

Code The code portions of the intermediate C0 machine are basically obtained by implementing the extended procedures by ordinary procedures and replacing every XCall by an ordinary procedure call.

- **Program rests.** If we replace in the program rest p_{x} of the extended machine each occurrence of a driver XCall by an invocation of the corresponding implementation function we get the program rest p_{i} of the intermediate machine.

  This replacement is defined by the following translation *trans*. Given a mapping $fns :: fun_n \to fun_n\ option$ from XCall names to function names, we define a translation of statements by:

  $$trans :: (fun_n \to fun_n\ option) \times stmt \to stmt$$

$$\begin{array}{lll}
\mathit{trans} & (\mathit{fns}, s_1;;s_2) & = \\
& \quad \mathit{trans}(\mathit{fns}, s_1);;\mathit{trans}(\mathit{fns}, s_2) & \\
& (\mathit{fns}, \mathsf{XCall}(\mathit{fn}_\mathrm{x}, \mathit{lvars}, \mathit{exps})) & = \\
& \quad \begin{cases} \mathsf{XCall}(\mathit{fn}_\mathrm{x}, \mathit{lvars}, \mathit{exps}) & : \mathit{fns}(\mathit{fn}_\mathrm{x}) = \bot \\ \mathsf{SCall}(\mathit{lvars}[0], & \\ \quad \mathsf{the}(\mathit{fns}(\mathit{fn}_\mathrm{x})), \mathit{exps}) & : \textbf{otherwise} \end{cases} & \\
& (\mathit{fns}, s) & = s
\end{array}$$

Note, that the translation only uses the first return value of an XCall and ignores all others. Thus, this restricts the set of XCalls that possibly can be translated.

Next, the relation between the program rests of the extended and the intermediate machine is defined by:

$$\mathit{xconsis}_\mathrm{prog}(\mathit{fns}, p_\mathrm{i}, p_\mathrm{x}) \equiv (\mathit{trans}(\mathit{fns}, p_\mathrm{x}) = p_\mathrm{i})$$

- **Procedure tables.** All functions defined in the procedure table pt_x of the extended machine must also be defined in the procedure table pt_i of the implementation machine. Additionally, the functions have to be equal except for their bodies for which the program rest relation from above has to hold.

  Formally we define the abstraction relation for procedure tables as follows:

$$\begin{array}{l}
\mathit{xconsis}_\mathrm{pt}(\mathit{fns}, pt_\mathrm{i}, pt_\mathrm{x}) \equiv \\
\quad \forall\, \mathit{fn}_\mathrm{x} \in \mathit{dom}(pt_\mathrm{x})\,. \\
\quad\quad pt_\mathrm{i}(\mathsf{the}(\mathit{fns}(\mathit{fn}_\mathrm{x}))) = (pt_\mathrm{x}(\mathit{fn}_\mathrm{x}))[\mathsf{body} := \mathit{trans}(\mathit{fns}, \mathit{fd}.\mathsf{body})]
\end{array}$$

- **Relating XCalls and C0 implementations.** This property ensures that the intermediate C0 machine implements the XCalls by corresponding C0 functions. The mapping between XCalls and their implementation is given by a so called *specification map*. A *specification map* maps the names of XCalls to simulate to (i) the signatures of the C0 functions implementing them, and (ii) the semantics of the XCalls. Formally, it has type $\mathit{specMap}_T = \mathit{fun}_n \rightarrow (\mathit{fun\text{-}decl}_T \times \mathit{xdecl}_T)\ \mathit{option}$

$$\mathit{xconsis}_\mathrm{xpt}(\mathit{fns}, \mathit{specMap}, pt_\mathrm{i}) \equiv$$

$$\begin{array}{ll}
\forall\, \mathit{fn}_\mathrm{x} \in \mathit{dom}(\mathit{fns}). & \mathit{specMap}(\mathit{fn}_\mathrm{x}) = \lfloor(\mathit{xdecl}, \mathit{fd}_\mathrm{i})\rfloor \wedge \\
& \mathit{xpt}(\mathit{fn}_\mathrm{x}) = \mathit{xdecl} \wedge \\
& pt_\mathrm{i}(\mathsf{the}(\mathit{fns}(\mathit{fn}_\mathrm{x}))) = \mathit{fd}_\mathrm{i}
\end{array}$$

The overall abstraction relation for the code can now be defined by:

$$\begin{aligned}
&\mathit{xconsis}_{\mathrm{code}}(\mathit{fns}, \mathit{specMap}, \mathit{pt}_{\mathrm{i}}, \mathit{pt}_{\mathrm{x}}, p_{\mathrm{i}}, p_{\mathrm{x}}) \equiv \\
&\qquad \mathit{xconsis}_{\mathrm{xpt}}(\mathit{fns}, \mathit{specMap}, \mathit{pt}_{\mathrm{i}}) \\
&\quad \wedge \quad \mathit{xconsis}_{\mathrm{prog}}(\mathit{fns}, p_{\mathrm{i}}, p_{\mathrm{x}}) \\
&\quad \wedge \quad \mathit{xconsis}_{\mathrm{pt}}(\mathit{fns}, \mathit{pt}_{\mathrm{i}}, \mathit{pt}_{\mathrm{x}})
\end{aligned}$$

Configurations The abstraction relation *xconsis* can now be defined. It takes as parameters the description of the intermediate C0 machine (consisting of type-environment, procedure table, and current state), the allocation function from the compiler, the current state of the VAMP assembly with devices model, the description of the extended C0 machine (consisting of procedure table, current state, and the extended procedure table) and two mappings, one for function names and one relating the XCall specifications to their implementation in the intermediate C0 machine. Moreover it is parametrized over the identifier of the device to abstract. It is made up of the following relations:

- **Extended C0 / VAMP assembly.** An extended machine can be used to abstract at most a single device id_X at a time. Note, that id_X is a parameter of the XCall correctness theorem (Theorem 9 stated below).

  The user (i.e. the one instantiating the XCall semantics) has to provide with an abstraction relation between the extended component on the one side, and the assembly memory and state of device id_X on the other side.

  $$\mathit{xconsis}_{\mathrm{ex}} :: \alpha \times ((\mathbb{N} \to \mathbb{Z}) \times S_{\mathfrak{D}}) \to \mathit{bool}$$

  However, not all abstractions are allowed. Memory portions which are covered by the intermediate C0 machine shall not be mapped to the extended component. Moreover, the abstracted device should be *stable* whenever consistency holds. Thus, given the last accessed memory address *max-address* of the C0 machine, we say the abstraction is valid if the following predicate holds (all free variables are assumed to be all quantified):

  $$\begin{aligned}
  &\mathit{valid}_{\mathrm{ex}}(\mathit{xconsis}_{\mathrm{ex}}, \mathit{max\text{-}address}) \equiv \\
  &\forall ad \leq \mathit{max\text{-}address}, v \in \mathbb{Z}\,. \\
  &\quad (\mathit{xconsis}_{\mathrm{ex}}(x, (\mathit{mm}, \mathit{dev})) \\
  &\quad\; \Longrightarrow \mathit{xconsis}_{\mathrm{ex}}(x, (\mathit{mm}(ad := v), \mathit{dev})) \wedge \mathit{stable}(\mathit{dev})).
  \end{aligned}$$

- **Extended C0 / Intermediate C0.** The memory of the intermediate C0 machine and the extended machine are equal. Additionally, the code and the procedure tables of the extended C0 machine and of the C0 implementation machine are related by the already defined relation $\mathit{xconsis}_{\mathrm{code}}$.

- **Intermediate C0 / VAMP assembly.** Compiler consistency holds between the intermediate C0 machine and the assembly machine.

Formally, we get for the overall abstraction relation:

$$\begin{aligned}
&\mathit{xconsis}(\mathit{tenv}, \mathit{pt}_{\mathrm{i}}, c, \mathit{alloc}, \mathit{asmd}, \mathit{pt}_{\mathrm{x}}, c_X, \mathit{xpt}, \mathit{fns}, \mathit{specMap}, \mathit{id}_X) \equiv \\
&\qquad \mathit{xconsis}_{\mathrm{ex}}(c_X.\mathsf{x}, (\mathit{asmd}.\mathsf{proc.mm}, \mathit{asmd}.\mathsf{devs}(\mathit{id}_X))) \\
&\wedge \quad c.\mathsf{mf} = c_X.\mathsf{c.mf} \\
&\wedge \quad \mathit{xconsis}_{\mathrm{code}}(\mathit{fns}, \mathit{specMap}, \mathit{pt}_{\mathrm{i}}, \mathit{pt}_{\mathrm{x}}, c.\mathsf{prog}, c_X.\mathsf{prog}) \\
&\wedge \quad \mathit{consis}(\mathit{alloc}, \mathit{tenv}, \mathit{pt}_{\mathrm{i}}, c, \mathit{asmd}.\mathsf{proc})
\end{aligned}$$

Correctness criteria of XCall

In this section we formalize the notion of implementation correctness of an XCall that is not controlling any device at all or only some device id_X. Basically, we require that the execution of a single XCall can be simulated by arbitrary many steps on VAMP assembly with devices and the intermediate C0 machine.

In the following we explain the required pre- and postconditions of the criteria:

- **Validity Predicates.** We have to assume validity of the C0 configuration, the extended C0 configuration, the assembly state and the execution sequences of the VAMP assembly with devices model.

  To apply C0 compiler correctness, validity of the initial C0 machine has to be assumed (see Section 3.3.7), i.e. the predicate *valid-C0* has to hold.

  Validity of the consistent assembly machine is provided by the predicate $\mathit{isa\text{-}asm\text{-}precond}_{\mathrm{init}}$.

  Since we assume that the underlying implementation of the XCall is controlling at most one device, correctness is proven only for those execution sequences which are restricted to steps of the processor and that device. Moreover these sequences should guarantee liveness of the processor and the devices. This is ensured by the restricted set of valid sequences $\mathit{Seq}_V(\mathit{id}_X)$. Note, that in case no device is accessed by the XCall, sequential assembly reasoning can be applied.

- **Translation from VAMP assembly to VAMP ISA.** The assembly model is only an abstraction (and simplification) of an underlying VAMP ISA. For translating computations from VAMP assembly with devices to VAMP ISA with devices via Theorem 1, we need to discharge a series of assumptions on the initial state and assumptions on each step of the execution. These are given by the already defined predicates $\mathit{isa\text{-}asm\text{-}precond}_{\mathrm{init}}$ and $\mathit{isa\text{-}asm\text{-}precond}_{\mathrm{dyn}}$ (cf. Section 3.1.5).

- **Memory restrictions.** The compiler correctness theorem is only applicable if in each step sufficient heap and stack memory is available in the assembly machine (see Section 3.3.7). For estimating memory consumption we apply the theory developed in Section 3.3.8. Note, that only claiming memory restrictions on the extended configuration does not work, since:

  - The procedure table of the implementation has more entries than the one of the extended machine: one more for each XCall. Hence, also the translated code size is larger.
  - On the extended machine, XCalls are executed for free, i.e. no stack or heap memory allocation is visible. However for its counterpart in the implementation, there is a *real* function call, with concrete memory requirements.

  For the heap memory, we can deduce that the intermediate machine will not consume more than the extended one. That follows, from the fact that C0 does not support any statements to free memory, and because both machines will finally have the same memory configuration again. Hence, by using Lemma 18 we only need to claim that enough heap is available for the extended configuration after executing the XCall. The heap size is bounded by the parameter $max\text{-}address$. By assuming $max\text{-}address$ not to be in the device domain, we ensure that the memory of the intermediate C0 machine is not intersecting the device domain.

  An upper bound of the stack consumption of functions with no recursive calls can be computed by a static analysis of the function bodies, as shown in Section 3.3.8. Remember, that this approach determines the deepest (and most stack consuming) path in the invocation tree of the given code. This deepest path is computed by the recursively defined set $ub\text{-}costs(tenv, pt, fn)$. An element of this set is an upper bound of the stack consumption required by the function fn in the procedure table pt.

  For the function to be simulated we get the size estimation sz by instantiating $ub\text{-}costs$ with the procedure table of the implementation machine and the name of the function implementing the XCall. We require that the sum of this upper bound and the current stack consumption (made up of the code size and the current stack size) of the intermediate C0 machine to be smaller than the heap-base.

- **Devices.** We assume that during executing the implementation no device at all is accessed or at most one, i.e that the computation is *pure*. This condition is necessary, to apply the C0 with inline assembly semantics developed in Section 3.4.

Let the function *specMap* define a specification and implementation of an XCall with name fn_{x} and abstraction relation $xconsis_{\mathrm{ex}}$. Moreover, we assume that the XCall implementation only controls the device id_X. We say the implementation is correct against the specification if the following predicate holds (note that all free variables in the definition are assumed to be all-quantified):

$$correct_{\mathrm{xcall}}(specMap, fn_{\mathrm{x}}, xconsis_{\mathrm{ex}}, id_X) \equiv$$

$$\begin{array}{ll}
 & valid_{\mathrm{ex}}(xconsis_{\mathrm{ex}}, max\text{-}address) \\
\wedge & valid\text{-}C0(tenv, pt_{\mathrm{i}}, c) \\
\wedge & isa\text{-}asm\text{-}precond_{\mathrm{init}}(asmd.\mathsf{proc}, code\text{-}base, code\text{-}size(tenv, pt_{\mathrm{i}}, c.\mathsf{mf}.\mathsf{gm})) \\
\wedge & \mathsf{heap\text{-}base} + heap\text{-}size(c_X'.\mathsf{c}, pt_{\mathrm{i}}) < max\text{-}address \\
\wedge & max\text{-}address < \mathsf{device\text{-}border} \\
\wedge & sz \in ub\text{-}costs(tenv, pt_{\mathrm{i}}, fns(fn_{\mathrm{x}})) \\
\wedge & sz + stack\text{-}start(tenv, pt_{\mathrm{i}}, c.\mathsf{mf}.\mathsf{gm}) + stack\text{-}size(tenv, c) \leq \mathsf{heap\text{-}base} \\
\wedge & xconsis(tenv, pt_{\mathrm{i}}, c, alloc, asmd, pt_{\mathrm{x}}, c_X, xpt, fns, specMap, id_X) \\
\wedge & c_X.\mathsf{c}.\mathsf{prog} = \mathsf{XCall}(fn_{\mathrm{x}}, lvars, exps_{\mathrm{p}}) \\
\wedge & \delta_{cx}(c_X, tenv, pt_{\mathrm{i}}, xpt) = \lfloor c_X' \rfloor \\
\Longrightarrow & (\forall seq \in Seq_V(id_X) \,.\, \exists c', alloc', T. \\
 & \quad xconsis(tenv, pt_{\mathrm{i}}, c', alloc', asmd^{seq,T}, pt_{\mathrm{x}}, c_X', xpt, \\
 & \quad\quad fns, specMap, id_X) \wedge \\
 & \quad isa\text{-}asm\text{-}precond_{\mathrm{dyn}}(asmd, code\text{-}base, \\
 & \quad\quad code\text{-}size(tenv, pt_{\mathrm{i}}, c.\mathsf{mf}.\mathsf{gm}), seq, T) \wedge \\
 & \quad isa\text{-}asm\text{-}precond_{\mathrm{init}}(asmd^{seq,T}.\mathsf{proc}, code\text{-}base, \\
 & \quad\quad code\text{-}size(tenv, pt_{\mathrm{i}}, c.\mathsf{mf}.\mathsf{gm})) \wedge \\
 & \quad valid\text{-}C0(tenv, pt_{\mathrm{i}}, c') \wedge \\
 & \quad pure(asmd, seq, T, id_X))
\end{array}$$

Now, we can extend compiler correctness to XCalls. The next theorem — under the assumption of correct implementation of all XCalls — establishes a simulation between an arbitrary execution of the extended machine on the one side (i.e. not only the execution of one XCall), and an intermediate C0 machine and the VAMP assembly with devices model on the other side. For the latter computation, all possible execution sequences are considered, (not only those restricted to some device). Note, that we propagate correctness at the level of function calls.

To formalize the theorem, we still have to describe two more properties:

- **Compositionality conditions.** The XCall correctness theorem serves as a translation of correctness results from the level of C0 with XCalls down to the VAMP assembly with devices model, in which the assembly state and devices are explicitly visible. Applied to the stack in the Acedemic System of Verisoft, the theorem is used to propagate the correctness of the page-fault handler (in which the hard disk driver is specified as XCall) to the level of VAMP assembly with devices. This result is then embedded into the verification of the kernel. However, the defined abstraction relation *xconsis* does not allow such an embedding of a smaller computation (page-fault handler) into a larger computation (kernel).

For such a transfer, the assumption on the program rest is too restrictive. It requires the code of the extended C0 machine and that of the implementation machine to be equal (up to XCalls substitution). This is a major restriction, as for example the page-fault handler is verified relative to an extended machine only containing the functions needed for its invocation not aware of the rest of the kernel code.

We generalize the abstraction relation, to obtain a modular verification chain:

- Since the extended machine is embedded into the implementation machine, the procedure table of the former must be included in the one of the latter. This is already granted by the $\mathit{xconsis}_{\mathrm{code}}$ relation.
- The program rest of the extended machine is only a prefix of the implementation machine (after XCall substitution). We define a prefix of a statement as follows:

$$\mathit{prefix}(p, p_{\mathrm{pre}}, \mathit{rest}) \equiv \mathit{s2l}(p) = \mathit{s2l}(p_{\mathrm{pre}})@\mathit{s2l}(\mathit{rest})$$

  Thus, we obtain the new abstraction relation *xconsis* by refining the code relation in the old abstraction relation as follows:

$$\begin{aligned}&\mathit{xconsis}(\mathit{tenv}, \mathit{pt}_{\mathrm{i}}, c, \mathit{alloc}, \mathit{asm}, \mathit{pt}_{\mathrm{x}}, c_X, \mathit{xpt}, \mathit{fns}, \mathit{specMap}, \mathit{rest}, \mathit{id}_X) =\\ &\quad \ldots\\ &\wedge \quad (\exists p_{\mathrm{pre}} \,.\, \mathit{prefix}(c.\mathsf{prog}, p_{\mathrm{pre}}, \mathit{rest}) \wedge\\ &\qquad\qquad \mathit{xconsis}_{\mathrm{code}}(\mathit{fns}, \mathit{specMap}, \mathit{pt}_{\mathrm{i}}, \mathit{pt}_{\mathrm{x}}, p_{\mathrm{pre}}, c_X.\mathsf{prog}))\end{aligned}$$

  Note, that the rest of the code, i.e. the part of the intermediate machine which is not simulated, stays unchanged during execution. At the end of the simulation the XCall machine will have processed its complete program, where the program rest of the intermediate machine still consists of the rest statement. This rest statement may now be verified without resorting to the XCall semantics (or by using some different XCall abstraction).

- **Validity of extended code.** Inline assembly code in the extended machine may break the abstraction provided by XCalls. For example, a code portion writing the abstracted memory region would have no semantical effects on the abstract memory component and hence lead to an unsound state. Therefore, we require the program rest of the extended machine and of the bodies of all functions possibly called during its execution to be free of any inline assembly code. Moreover, we require that only XCalls defined in the specification map may be invoked. To formulate these dynamic properties on the program rest we use the predicate $\mathit{valid\text{-}prop}_{\mathrm{code}}$ and the formalism developed in Section 3.3.6.

Now, the simulation theorem reads in its full beauty as follows:

Theorem 9 (C0 with XCalls compiler correctness)

$$
\begin{array}{ll}
 & (\forall \mathit{fn}_x \in \mathit{dom}(\mathit{specMap}) \,.\, \mathit{correct}_{\mathit{xcall}}(\mathit{specMap}, \mathit{fn}_x, \mathit{xconsis}_{\mathit{ex}}, \mathit{id}_X)) \\
\wedge & \mathit{valid\text{-}C0}(\mathit{tenv}, \mathit{pt}_i, c) \\
\wedge & \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asmd}.\mathsf{proc}, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm})) \\
\wedge & \mathsf{heap\text{-}base} + \mathit{heap\text{-}size}(c_X{}'.\mathsf{c}, \mathit{pt}_i) < \mathit{max\text{-}address} \\
\wedge & \mathit{max\text{-}address} < \mathsf{device\text{-}border} \\
\wedge & \mathit{sz} \in \mathit{ub\text{-}costs}(\mathit{tenv}, \mathit{pt}_i, \mathit{fn}_i) \\
\wedge & \mathit{sz} + \mathit{stack\text{-}start}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm}) + \mathit{stack\text{-}size}(\mathit{tenv}, c) \leq \mathsf{heap\text{-}base} \\
\wedge & \mathit{valid\text{-}prop}_{\mathit{code}}(c_X.\mathsf{c}.\mathsf{prog}, \mathit{pt}_x, (\lambda s.s \neq \mathsf{Asm}(\dots))) \\
\wedge & \mathit{valid\text{-}prop}_{\mathit{code}}(c_X.\mathsf{c}.\mathsf{prog}, \mathit{pt}_x, \\
 & \qquad (\lambda s.s = \mathsf{XCall}(\mathit{fn}_x, \dots) \implies \mathit{fn}_x \in \mathit{dom}(\mathit{specMap}))) \\
\wedge & \mathit{xconsis}(\mathit{tenv}, \mathit{pt}_i, c, \mathit{alloc}, \mathit{asmd}, \mathit{pt}_x, c_X, \mathit{xpt}, \mathit{fns}, \mathit{specMap}, \mathit{rest}, \mathit{id}_X) \\
\wedge & c_X.\mathsf{c}.\mathsf{prog} = \mathsf{SCall}(\dots, \mathit{fn}_i, \dots) \\
\wedge & \delta_{cx}{}^k(c_X, \mathit{tenv}, \mathit{pt}_i, \mathit{xpt}) = \lfloor c_X{}' \rfloor \\
\implies & (\forall \mathit{seq} \in \mathit{Seq}_V \,.\, \exists c', \mathit{alloc}', T. \\
 & \qquad \mathit{xconsis}(\mathit{tenv}, \mathit{pt}_i, c', \mathit{alloc}', \mathit{asmd}^{\mathit{seq},T}, \mathit{pt}_x, c_X{}', \mathit{xpt}, \\
 & \qquad\qquad \mathit{fns}, \mathit{specMap}, \mathit{rest}, \mathit{id}_X) \wedge \\
 & \qquad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{dyn}}(\mathit{asmd}, \mathit{code\text{-}base}, \\
 & \qquad\qquad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm}), \mathit{seq}, T) \wedge \\
 & \qquad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asmd}^{\mathit{seq},T}.\mathsf{proc}, \mathit{code\text{-}base}, \\
 & \qquad\qquad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm})) \wedge \\
 & \qquad \mathit{valid\text{-}C0}(\mathit{tenv}, \mathit{pt}_i, c') \wedge \\
 & \qquad \mathit{pure}(\mathit{asmd}, \mathit{seq}, T, \mathit{id}_X))
\end{array}
$$

Proof. The theorem is proven by induction on k. The induction base is trivially true by choosing $c' = c$, $\mathit{alloc}' = \mathit{alloc}$ and $T = 0$.

Next, we consider the induction step for $k+1$ (variables of the induction hypothesis are marked by a superscript h). We instantiate the induction hypothesis with $k^h = k$. Out of the obtained premises all but the condition on enough heap follow straight from the premises of the theorem. Enough heap is ensured since the heap memory consumption is monotonically increasing (Lemma 18). Thus, we can assume that all postconditions of the theorem hold after k steps. We proceed with a case distinction on the head of the program rest of the extended machine after k steps (denoted with $c_X{}^k$). For

$hd(s2l({c_X}^k.\mathsf{c.prog}))$ we have two cases, either the next statement is not an XCall invocation — or it is:

- **Ordinary C0 statement** We can conclude that in the first case the head of the program rest is not an inline assembly statement. This follows from the first condition on the program rest of the extended machine and from code consistency. We execute an ordinary C0 step of the intermediate machine. Extended consistency is established as follows: By applying extended compiler correctness (Theorem 5, or alternatively by applying the first rule of the $\rightarrow_{\mathrm{c+ad}}$ semantics) we obtain a consistent new assembly machine. All assumptions of compiler consistency are trivially ensured, except for the assumption on stack memory consumption. This one is deduced by our static estimation of stack consumption and by Theorem 6.

  Consistency on the extended component is established by proving that neither the device state nor the memory region being abstracted have been changed in the meanwhile. The first follows from an application of Lemma 6. The latter is verified using the validity of the extended mapping and by Lemma 16 ensuring no access beyond *max-address*.

- **Invocation to an XCall** First, we show that the invoked XCall is indeed defined in the *specMap*. This follows from the second condition on the code of the extended machine and by Lemma 14. We conclude from the first premise of the theorem that the XCall is implemented correctly. Overall correctness of the theorem follows almost straight from implementation correctness. It remains to generalize the reduced sequences to arbitrary ones. Lemma 6 together with the consistency assumption of stable device state proves this.

q.e.d.

Note, that one could easily prove a stronger version of the theorem which neither requires stability of the device to abstract nor is restricted to the abstraction of a single device. Such a theorem would be proven using the semantics introduced in Section 3.4.

Embedding XCalls into Hoare logic

In this thesis we omit the description of XCall embedding into higher levels of the C0 language stack. A detailed description of an extended big-step semantics and Hoare logic can be found in [AHL$^+$09] (conducted by N. Schirmer). In the following we briefly sketch the extension to Hoare logic.

We apply an instance of Schirmer's Hoare logic environment [Sch06] implemented in Isabelle/HOL. He defines a set of Hoare rules, for a generic programming language *Simpl*, and formally proves the soundness of this logic. Hoare rules describe a triple $\{P\}S\{Q\}$, where precondition P defines the set of valid initial states of the C0 variables, S is the statement to execute and postcondition Q is guaranteed to hold for the states after execution of S. Schirmer's formalization in Isabelle/HOL guarantees even total correctness and hence termination.

Additionally to the embedding of C0 into Simpl, some special treatment for XCalls is introduced. We refer to the resulting system as *extended Hoare logic*. In Simpl the extended state is treated analogous to ordinary C0 variables, with the only difference that they are not restricted to C0 types. It is not necessary to introduce a new Hoare rule into Simpl to handle XCalls. Instead we can use the *Basic* construct of Simpl, a general assignment which can deal with an arbitrary state update function f: $\{f(s) \in Q\}$ *Basic* f $\{Q\}$. On the level of Simpl every XCall is modelled as such a state update, representing the abstract semantics of the XCall.

Next, we apply the developed language stack to verify the correctness of a hard disk driver in the context of a page-fault handler of a micro kernel.

Chapter 4

Case Study: Verifying a Hard Disk Driver in the Context of Memory Virtualization

One of the most challenging parts of verifying the generic Verisoft microkernel CVM is memory virtualization, i.e. to ensure that each user process controls its own, large, and isolated memory. User processes access memory by virtual addresses, which are subsequently translated to physical ones. Modern computer systems implement virtual memory by demand paging: small consecutive chunks of data, called pages, are either stored in a fast but small physical memory or in a large but slower swap memory, which is usually implemented by a hard disk. The page table, a data structure both accessed by the processor and by software, maintains whether a page is in the swap or the physical memory. A process attempting to access a page currently in swap memory causes the processor to signal a page-fault interrupt. On the hardware side, the memory management unit (MMU) triggers the interrupt and translates from virtual to physical page addresses. On the software side, the *page-fault handler* reacts to page-faults by loading the requested page to the physical memory. If the physical memory is full, some other page is swapped out.

The swap component is a hard disk and the corresponding swap operations are driver write and read calls to that disk. Each of these calls is written in assembly and encapsulated into a C0 function. The page-fault handler is implemented almost completely in C0, except for the invoked hard disk driver which accesses the devices and portions of the memory not covered by the kernel C0 machine. The page-fault handler itself is a C0 function invoked by the kernel machine.

The correctness of the page-fault handler is ultimately expressed at the level of VAMP ISA with devices and embedded into the larger computation of the kernel code [ASS08]. To avoid verifying the whole page-fault handler at the level of VAMP ISA with devices we leverage the extended language stack developed before. First the driver semantics is expressed in terms of XCalls. These XCalls are used to verify the code of the page-fault handler in the C0 Hoare logic which is extended by a general assignment rule covering XCall semantics. The obtained results are then transferred to the extended C0 small step semantics, and subsequently lifted via a correctness proof of the implementation to the level of VAMP ISA with devices.

We start this section with a definition of an accurate hard disk model. Next, we introduce the assembly portion of the driver (write case) and prove its correctness [AH08]. This driver is embedded into a C0 function call and specified in terms of XCall semantics. Finally, we prove the implementation correctness of the driver write XCall and the theorem which is used to transfer the page-fault handler from the level of C0 with XCalls down to the level VAMP ISA with devices [AHL+09].

Note, that our final theorem has been successfully applied in Isabelle/HOL to verify the the page-fault handler [Sta09] and the kernel [Tsy09].

4.1 Hard Disk Model

Our formal hard disk controller model is based on a subset of the ATAPI standard [Ame00]. We restrict ourselves to a subset of the commands and assume that only a single disk is hooked up to the controller. Note that while such controllers usually allow to access two hard disks called the master and the slave disk, we will in the following assume that only the master disk is accessed and connected. Hence, we use the terms disk and controller interchangeably.

4.1.1 Overview

Hard disks are parameterized over the number of sectors $0 < S \leq 2^{28}$ they store. Each sector stores 128 words, making up for a content of up to $S \cdot 2^9 \leq$ 128 GB. The disk is accessed by issuing commands to it; we only model three of them. The *reset* command takes the disk into a well-defined initial state at any time. The *read* and *write* commands load and store a range of sectors. This range is processed sector by sector, with each sector first being transferred into an internal (volatile) buffer of the disk and then to the processor resp. the disk. The processor may access the internal buffer in a sequential fashion by performing 128 word operations (reads or writes) on the *data port* of the disk. In *polling mode*, the processor must query the disk for the completion of a sector transfer; in *interrupt mode*, the disk causes an interrupt for each sector transferred from resp. to the disk.

4.1.2 Addressing Sectors

In an ATA/ATAPI compatible hard disk sectors are addressed via 28-bit (logical) block addresses, which are composed of four parts, from low to high order: an 8-bit sector number, an 8-bit cylinder-low, an 8-bit cylinder-high, and a 4-bit head. In the formal model we represent an *lba* as element of the record type lba_T with the following four fields:

$\mathsf{snumb} : \mathbb{N}$	The sector number.
$\mathsf{cyll} : \mathbb{N}$	The cylinder low.
$\mathsf{cylh} : \mathbb{N}$	The cylinder high.
$\mathsf{drivehead} : \mathbb{N}$	The drive head.

This representation is chosen, since sector addresses are past component by component through corresponding ports to the hard disk.[1]

The translation from logical block addresses to sector numbers is straightforward:

$$\textit{lba-to-nat} :: lba_T \rightarrow \mathbb{N}$$

$$\begin{aligned}&\textit{lba-to-nat}(ba) \equiv\\ &ba.\mathsf{snumb} + (ba.\mathsf{cyll} + (ba.\mathsf{cylh} + ba.\mathsf{drivehead} \cdot 256) \cdot 256) \cdot 256\end{aligned}$$

Next, we define an increment operator on logical block addresses. Obviously, this operator should be consistent with ordinary incrementation (modulo 2^{28}) of the *lba* interpreted as natural. Formally, incrementation is defined by the following two functions:

$$\textit{inc-ba-block} :: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\textit{inc-ba-block}(cin, a) \equiv ((cin + a) \div 256, (cin + a) \mod 256)$$

$$\textit{inc-lba} :: lba_T \rightarrow lba_T$$

```
$\textit{inc-lba}(lba) \equiv$
 let
   $(c_0, \textit{snumb}') = \textit{inc-lba-block}(1, lba.\mathsf{snumb})$
   $(c_1, \textit{cyll}') = \textit{inc-lba-block}(c_0, lba.\mathsf{cyll})$
   $(c_2, \textit{cylh}') = \textit{inc-lba-block}(c_1, lba.\mathsf{cylh})$
   $\textit{drivehead}' = \textit{inc-lba-block}(c_2, lba.\mathsf{drivehead})$
 in
   $(\mathsf{snumb} = \textit{snumb}', \mathsf{cyll} = \textit{cyll}', \mathsf{cylh} = \textit{cylh}', \mathsf{drivehead} = \textit{drivehead}')$
```

[1]For *lba* addresses, the components do no longer have any geometric meaning on the physical hardware.

4.1.3 Configuration

Now, we can define the configuration of a hard disk. It is of record type C_{hd} with the following fields:

$\mathsf{sm} : \mathbb{N} \to \mathbb{N}$	The sector memory is a word addressable memory of words (32-bit binary number interpretation of entries). It stores the disk content.
$\mathsf{buf} : \mathbb{N} \to \mathbb{N}$	The sector buffer, which is a word buffer (32-bit binary number interpretation of entries) and is accessed sequentially.
$\mathsf{bp} : \mathbb{N}$	The (word) buffer pointer.
$\mathsf{lba} : lba_T$	The current sector address.
$\mathsf{scnt} : \mathbb{N}$	The number of sectors to process.
$\mathsf{ien} : bool$	The interrupt enable flag, which is 0 for polling mode.
$\mathsf{int} : \mathbb{N}$	The pending interrupt flag.
$\mathsf{cs} \in Hd_{cs}$	The control state, where the set of states is defined as $Hd_{cs} = \{\mathsf{idle}, \mathsf{brd}, \mathsf{bwr}, \mathsf{prd}, \mathsf{pwr}, \mathsf{err}\}$. In idle state, the disk is ready to take new commands. Read commands will repeatedly visit the states *brd* and *prd*. In the former the disk fills its buffer, which is then read out by the processor. Likewise, write commands visit the states pwr and bwr. In the former the processor fills the disk's buffer, which is then written to the disk. The state err is entered when invalid commands are issued to the disk.

Not all configuration encode valid abstractions of real hard disk states. As for assembly configurations, we have to restrict the set of disk states, to valid ones. These restrictions are: (i) The logical base address is in range, (ii) the buffer pointer is pointing to a valid buffer position, i.e. it is smaller than 128, (iii) the sector memory is well-typed, i.e., each entry of sm is in the range of 32-bit binary numbers, and (iv) the buffer is well-typed, i.e., each entry of buf is in the range of 32-bit binary numbers. Formally we get:

$$valid\text{-}disk :: C_{hd} \times bool$$

$$\begin{aligned} &valid\text{-}disk(c_{hd}) \equiv \\ &\quad c_{hd}.\mathsf{bp} < 128 \\ &\wedge \quad lba\text{-}to\text{-}nat(c_{hd}.\mathsf{lba}) < S \\ &\wedge \quad \forall i < S \cdot 128 \,.\, asm\text{-}nat(c_{hd}.\mathsf{sm}(i)) \\ &\wedge \quad \forall i < 128 \,.\, asm\text{-}nat(c_{hd}.\mathsf{buf}(i)) \end{aligned}$$

Note, that for the parameter S we have implicitly the two restrictions: $S > 0$ and $S < 2^{28}$.

Initial disk configurations, are configurations which are valid, in idle state and in which the buffer pointer is zero.

$$\textit{initial-disk} :: C_{hd} \times \textit{bool}$$

$$\begin{aligned} &\textit{initial-disk}(c_{hd}) \equiv \\ &\quad \textit{valid-disk}(c_{hd}) \\ &\wedge \quad c_{hd}.\mathsf{cs} = \mathsf{idle} \\ &\wedge \quad c_{hd}.\mathsf{bp} = 0 \end{aligned}$$

The interrupt function is equal to the corresponding component in the configuration, i.e. $ir_{\mathrm{hd}}(c_{hd}) \equiv c_{hd}.\mathsf{int}$.

4.1.4 Transitions

Disk computation is modeled by a transition function δ_{hd}. It takes an input *eifi* from an external environment, an input *mifi* from the processor, and the disk's current state c_{hd}. It returns an updated state c_{hd}' and an output *mifo* to the processor. The external output is always empty, i.e. $\textit{eifo} = \textit{eifo}_\epsilon$.

The time it takes the disk to copy data from the buffer to the sector memory and vice versa is modeled by non-deterministic input from the external environment. The external input is a trigger, indicating the end of the current buffer transfer if set to true. We have:

$$\textit{Eifi}_{\mathrm{hd}} = \textit{bool}$$

In the following we first describe the ports of the hard disk, identify which port accesses are considered illegal, define the control state transitions and the semantics of legal port accesses, and finally formalize the semantics of external steps of the hard disk.

Ports The hard disk provides access to six ports (cf. Table 4.1): (i) Commands are written to the port $\mathsf{cmd}_{\mathrm{p}}$, (ii) the ports $\mathsf{snumb}_{\mathrm{p}}$, $\mathsf{cyll}_{\mathrm{p}}$, $\mathsf{cylh}_{\mathrm{p}}$, and $\mathsf{drivehead}_{\mathrm{p}}$ are used to pass sector addresses, (iii) the count of sectors still to process is accessed through the port $\mathsf{scnt}_{\mathrm{p}}$, (iv) the disk buffer is accessed through the port $\mathsf{data}_{\mathrm{p}}$, and (v) the current status of a pending operation can be read from the port $\mathsf{status}_{\mathrm{p}}$ (if polling it provides value 1 other 0).

For defining the transition function, we first introduce the following abbreviations on port accesses: we write $rd(\textit{mifi}, \textit{port})$ for $\textit{mifi}.\mathsf{ad} = \textit{port} \wedge \textit{mifi}.\mathsf{rd} \equiv \mathsf{true}$ and $wr(\textit{mifi}, \textit{port})$ for $\textit{mifi}.\mathsf{ad} \equiv \textit{port} \wedge \textit{mifi}.\mathsf{wr} = \mathsf{true}$. Moreover, commands issued to the hard disk are abbreviated by $\textit{cmd}(\textit{mifi}, c) \equiv wr(\textit{mifi}, \mathsf{cmd}_{\mathrm{p}}) \wedge \textit{mifi}.\mathsf{din} = c$.

In the formal model we confine ourselves to only three commands: (i) reset, denoted by $\textit{cmd}(\textit{mifi}, \mathsf{RST}_{\mathrm{c}})$, (ii) write, denoted by $\textit{cmd}(\textit{mifi}, \mathsf{WR}_{\mathrm{c}})$, and (iii) read, denoted by $\textit{cmd}(\textit{mifi}, \mathsf{RD}_{\mathrm{c}})$,

Port	Access	Abbreviation	Name	x86 Port
0	R/W	data_p	Data	`0x1f0`
2	R/W	scnt_p	Sector count	`0x1f2`
3	R/W	$\mathsf{snumb}_\mathrm{p}$	Sector number	`0x1f3`
4	R/W	cyll_p	Cylinder low	`0x1f4`
5	R/W	cylh_p	Cylinder high	`0x1f5`
6	R/W	$\mathsf{drivehead}_\mathrm{p}$	Drive / head	`0x1f6`
7	W	cmd_p	Command	`0x1f7`
7	R	stat_p	Status	`0x1f7`
8	W	$\mathsf{devicecontrol}_\mathrm{p}$	Device control	`0x3f6`

Table 4.1: Ports of the Hard Disk (the last column lists the corresponding default port numbers for the x86 architecture; this information is not used for the formal specification and verification.)

Issuing the reset command, $cmd(\mathit{mifi}, \mathsf{RST}_\mathrm{c})$, has top priority: the disk enters the idle state, $c_{hd}'.\mathsf{cs} = \mathsf{idle}$, and the buffer pointer, the interrupt enable flag, and the pending interrupt flag are set to zero. The other components do not change; the value of *mifo* is irrelevant (for any write).

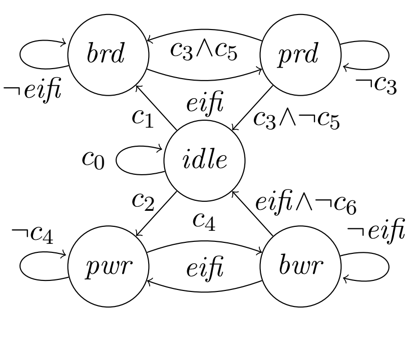
If no reset command is issued, the hard disk either makes an error transition, setting $c_{hd}'.\mathsf{cs} = \mathsf{err}$ and $\mathit{mifo} = 0$, or a regular transition. Ignoring hardware errors, we have two conditions that trigger error transitions: illegal port accesses or absence of enabled regular transitions. Absence of error transitions should guarantee that the processor handles the device correctly.

Illegal port accesses The function *illegal-access* denotes for a given *mifi* illegal accesses.

$$\mathit{illegal\text{-}access} :: \mathit{Mifi}_T \to \mathit{bool}$$

$$\mathit{illegal\text{-}access}(\mathit{mifi}) \equiv$$

$req(\mathit{mifi}, a)$	An access to a non-modeled or unused port $a \in \{1, 9, 10, \ldots, 1023\}$.
$rd(\mathit{mifi}, \mathsf{devicecontrol}_\mathrm{p})$	A device control port read (usually interpreted as read to alternate status port, which we do not model here.)
$rd(\mathit{mifi}, \mathsf{stat}_\mathrm{p}) \wedge c_{hd}.\mathsf{ien}$ $\wedge c_{hd}.\mathsf{cs} \in \{\mathsf{brd}, \mathsf{bwr}\}$	A status port read if interrupts are enabled and the disk is in buffer read or write state.
$rd(\mathit{mifi}, \mathsf{data}_\mathrm{p}) \wedge c_{hd}.\mathsf{cs} \neq \mathsf{prd}$	A data port read in a state other than processor read.



$c_0 = \neg wr(mifi, cmd_{\mathrm{p}})$
$\hat{c} = c_{hd}.\mathsf{scnt} \neq 0 \wedge (S \geq c_{hd}.\mathsf{lba} + c_{hd}.\mathsf{scnt})$
$c_1 = cmd(mifi, rd_{\mathrm{c}}) \wedge \hat{c}$
$c_2 = cmd(mifi, wr_{\mathrm{c}}) \wedge \hat{c}$
$\tilde{c} = (c_{hd}.\mathsf{bp} = 127)$
$c_3 = rd(mifi, data_{\mathrm{p}}) \wedge \tilde{c}$
$c_4 = wr(mifi, data_{\mathrm{p}}) \wedge \tilde{c}$
$c_5 = c_{hd}.\mathsf{scnt} > 0$
$c_6 = c_{hd}.\mathsf{scnt} > 1$

Figure 4.1: Regular state transitions

$wr(mifi, \mathsf{data}_{\mathrm{p}}) \wedge c_{hd}.\mathsf{cs} \neq \mathsf{pwr}$	A data port write in a state other than processor write.
$wr(mifi, x) \wedge (x \neq \mathsf{data}_{\mathrm{p}})$ $\wedge mifi.\mathsf{din} > 255$	A write to a port other than the data port with a value greater than 255.
$cmd(mifi, c) \wedge c \notin \{\mathsf{rst}_{\mathrm{c}}, \mathsf{rd}_{\mathrm{c}}, \mathsf{wr}_{\mathrm{c}}\}$	A write of an non-modeled or unknown command to the command port.
$wr(mifi, \mathsf{drivehead}_{\mathrm{p}})$ $\wedge mifi.\mathsf{din} \div 16 \neq 14$	Drive/head port reads must have bits [7:0] equal to $0b1110 = to\text{-}bin(14)$, which select logical block addressing and the master disk. The slave disk is currently not modeled.
$wr(mifi, \mathsf{devicecontrol}_{\mathrm{p}})$ $\wedge mifi.\mathsf{din} \notin \{0, 2\}$	An access to ports for command parameters while the state is not idle. A write different from 0 or 2 to the device control port.
$req(mifi, a) \wedge c_{hd}.\mathsf{cs} \neq \mathsf{idle}$	An access to ports for command parameters while the state is not idle, i.e. for $a \in \{\mathsf{scnt}_{\mathrm{p}}, \mathsf{snumb}_{\mathrm{p}}, \mathsf{cyll}_{\mathrm{p}}, \mathsf{cylh}_{\mathrm{p}}, \mathsf{drivehead}_{\mathrm{p}}, \mathsf{devicecontrol}_{\mathrm{p}}\}$.

Regular State transitions We denote the regular (control) state transitions by the function

$$\delta_{\mathrm{cs}} :: C_{hd} \times Mifi_T \times Eifi_{\mathrm{hd}} \rightarrow Hd_{\mathrm{cs}}$$

The function is defined by the automaton in Figure 4.1. Edges are labeled with transition constraints. All transitions not depicted in the figure, are assumed to lead to the error state, err.

Thus, for all transitions (except in case of a reset) of the hard disk the next control state is computed as follows:

$$c_{hd}.\mathsf{cs} = \begin{array}{l} \textbf{if } \mathit{illegal\text{-}access}(\mathit{mifi})\textbf{then} \\ \quad \mathsf{err} \\ \textbf{else} \\ \quad \delta_{\mathrm{cs}}(c_{hd}, \mathit{mifi}, \mathit{eifi}) \end{array}$$

Next, we describe the effects of regular transitions for legal port accesses. Components which are not mentioned are assumed to stay unchanged. A command port write is fully described by the transition of the control automaton. Accesses to other ports are defined as follows:

- Accesses to ports x_{p} for $x \in \{\mathsf{snumb}, \mathsf{cyll}, \mathsf{cylh}, \mathsf{drivehead}\}$ operate on one of the four components of the (current) sector address $c_{hd}.\mathsf{lba}$.

  For reads, i.e., $rd(\mathit{mifi}, x_{\mathrm{p}})$, we set $c_{hd}{}' = c_{hd}$. The data returned to the processor is defined as follows:

  $$\mathit{mifo} = \begin{cases} c_{hd}.\mathsf{lba.snumb} & \text{if } x = \mathsf{snumb} \\ c_{hd}.\mathsf{lba.cyll} & \text{if } x = \mathsf{cyll} \\ c_{hd}.\mathsf{lba.cylh} & \text{if } x = \mathsf{cylh} \\ 224 + c_{hd}.\mathsf{lba.drivehead} & \text{if } x = \mathsf{drivehead} \end{cases}$$

  For writes, i.e., $wr(\mathit{mifi}, x_{\mathrm{p}})$ we update the $c_{hd}.\mathit{lba}$ component. If $x = \mathsf{snumb}$, we set $c_{hd}{}'.\mathsf{lba.snumb} = \mathit{mifi}.\mathsf{din}$. If $x = \mathsf{cyll}$, we set $c_{hd}{}'.\mathsf{lba.cyll} = \mathit{mifi}.\mathsf{din}$. If $x = \mathsf{cylh}$, we set $c_{hd}{}'.\mathsf{lba.cylh} = \mathit{mifi}.\mathsf{din}$. If $x = \mathsf{drivehead}$, we set $c_{hd}{}'.\mathsf{lba.drivehead} = \mathit{mifi}.\mathsf{din}$.

- An access to port $\mathsf{scnt}_{\mathrm{p}}$ operates on the component $c_{hd}.\mathsf{scnt}$.

  For reads, i.e., $rd(\mathit{mifi}, \mathsf{scnt}_{\mathrm{p}})$, we set $c_{hd}{}' = c_{hd}$ and return $\mathit{mifo} = c_{hd}.\mathsf{scnt}$.

  For writes, i.e., $wr(\mathit{mifi}, \mathsf{scnt}_{\mathrm{p}})$, we set $c_{hd}{}'.\mathsf{scnt}$ to $\mathit{mifi}.\mathsf{din}$ if $\mathit{mifi}.\mathsf{din} \neq 0$ and to 256 otherwise. Thus, an input value of 0 encodes a sector count of 256 and sector count writes yield a non-zero sector count component by definition.

- Access to the device control port $\mathsf{devicecontrol}_{\mathrm{p}}$ allow to read and update the interrupt enable flag $c_{hd}.\mathsf{ien}$. Note that regularly the device control port also allows to issue software reset to the hard disk [Ame00]. We do not model these resets.

  Reads to the ports are modeled as illegal operations.

For writes, i.e., $wr(mifi, \mathsf{devicecontrol_p})$, we set $c_{hd}'.\mathsf{ien} = (((mifi.\mathsf{din} \bmod 4) \div 2) = 0)$.

- A legal access to the data port $\mathsf{data_p}$ has memory semantics and operates on the current word $c_{hd}.\mathsf{buf}(c_{hd}.\mathsf{bp})$ of the buffer. Hence, in case of a read operation, $wr(mifi, \mathsf{data_p})$, the current word $c_{hd}.\mathsf{buf}(c_{hd}.\mathsf{bp})$ is returned. In case of a write operation, $rd(mifi, \mathsf{data_p})$, the received word is stored in the buffer, $c_{hd}'.\mathsf{buf}(c_{hd}.\mathsf{bp} := mifi.\mathsf{din})$.

  As a side effect, a buffer access (even a read) increments the buffer pointer: $c_{hd}'.\mathsf{bp} = c_{hd}.\mathsf{bp} + 1 \mod 128$.

- A status port read, $rd(mifi, \mathsf{stat_p})$, returns 128 if the disk is still transfering data between buffer and sector memory, i.e. if $c_{hd}.\mathsf{cs} \in \{\mathsf{brd}, \mathsf{bwr}\}$ and 0 otherwise. As a side effect, such a read also clears the pending interrupt flag, $c_{hd}'.\mathsf{int} = 0$.

External transitions The effect of the external input on the control state is described by Figure 4.1.

Moreover, in case the hard disk receives an external trigger while it is currently in a buffer read or write state, the start sector is incremented and the sector count is decremented. For buffer write, the buffer contents are copied to the disk contents. For buffer read, the requested sector is copied from the sector memory to the buffer. The pending interrupt flag is turned on in interrupt mode.

Before presenting the formal definition of an external transition, we introduce the following function, which denotes a copy operation of l words starting at address s in memory m to memory m' starting at address s'.

$$\begin{aligned} mcopy \quad & (m, m', s, s', 0) && = m \\ & (m, m', s, s', (l+1)) && = mcopy(m(s + l := m'(s' + l)), m', s, s', l) \end{aligned}$$

We abbreviate the application of $mcopy$ by writing $m_l(s) := m'(s')$ instead of $memcopy(m, m', s, s', l)$.

An external transition $\delta_{\mathrm{hd}}(c_{hd}, mifi_\epsilon, eifi)$ returns the triple $(c_{hd}', mifo_\epsilon, eifo_\epsilon)$, where the next state ${c_{hd}}'$ is defined as follows:

```
$c_{hd}[$
lba    $:= \mathit{inc\text{-}lba}(c_{hd}.\mathsf{lba}),$
scnt   $:= c_{hd}.\mathsf{scnt} - 1,$
int    $:= c_{hd}.\mathsf{ien},$
sm     $:=$ if $c_{hd}.\mathsf{cs} = \mathsf{bwr}$ then
              $c_{hd}.\mathsf{sm}_{128}(\mathit{lba\text{-}to\text{-}nat}(c_{hd}.\mathsf{lba}) \cdot 128) := c_{hd}.\mathsf{buf}(0),$
           else
              $c_{hd}.\mathsf{sm}$
buf    $:=$ if $c_{hd}.\mathsf{cs} = \mathsf{brd}$ then
              $c_{hd}.\mathsf{buf}_{128}(0) := c_{hd}.\mathsf{sm}(\mathit{lba\text{-}to\text{-}nat}(c_{hd}.\mathsf{lba}) \cdot 128),$
           else
              $c_{hd}.\mathsf{buf}]$
```

Note, that for liveness reasons, the trigger must be active infinitely often. That means, we have to restrict the set Seq_V of valid execution sequences by the following additional condition:

$$\mathit{hd\text{-}trigger\text{-}live} :: \mathit{Seq}_T \to \mathit{bool}$$

$$\mathit{hd\text{-}trigger\text{-}live}(\mathit{seq}) \equiv \mathit{live}(\mathit{seq}, \lambda \mathit{ev}.\mathit{ev} = (\mathit{id}_{\mathrm{hd}}, \mathsf{true}))$$

This restriction is justified by the hardware implementation of the hard disk. Formally, one has to prove that the the property holds for the scheduling function sI_{PD} of the simulation theorem between ISA and gate-level, i.e. that $\mathit{range}(sI_{PD}) \subseteq \mathit{Seq}_V$ holds.

Moreover, *hd-trigger-live* is a separable environment restriction (cf. Sect. 3.2.1).

Lemma 24 (Observation Valid on Sequence Set)

$$\mathit{valid\text{-}ob}(\mathit{Seq}_V)$$

```
Ilw 15 topl sa_off    (0.1)    Ixori 1 0 D0(id_hd)         (1.1)   ┌─→Iaddi 10 0 128  (2.1)
Ilw 2 topl ma_off     (0.2)    Isw 4 1 devicecontrol_p     (1.2)   │┌→Ilw 3 2 0        (3.1)
Iandi 16 15 255       (0.3)    Isw 12 1 scnt_p             (1.3)   ││ Isw 3 1 data_p   (3.2)
Isli 17 15 8          (0.4)    Isw 16 1 snumb_p            (1.4)   ││ Isubi 10 10 1    (3.3)
Iandi 17 17 255       (0.5)    Isw 17 1 cyll_p             (1.5)   ││ Ibnez 10 -16     (3.4)
Isli 18 15 16         (0.6)    Isw 18 1 cylh_p             (1.6)   │└─Iaddi 2 2 4      (3.5)
Iandi 18 18 255       (0.7)    Isw 19 drivehead_p          (1.7)   │┌→Ilw 3 1 stat_p   (4.1)
Isli 19 15 24         (0.8)    Isw 3 1 cmd_p               (1.8)   ││ Isgei 14 3 128   (4.2)
Iandi 19 19 15        (0.9)                                        ││ Ibnez 14 -12     (4.3)
Iaddi 19 19 224       (0.10)                                       │└─nop              (4.4)
Iaddi 3 0 wr_c        (0.11)                                       │  Isubi 12 12 1    (5.1)
Iaddi 4 0 2           (0.12)                                       │  Ibnez 12 -48     (5.2)
Iaddi 12 0 8          (0.13)                                       └──nop              (5.3)
```

Figure 4.2: Device Driver — $d\text{-}code(id_{\text{hd}}, sa_{\text{off}}, ma_{\text{off}})$

4.2 Assembly Driver

We present a assembly driver for the hard disk for which we have formally proven correctness in Isabelle/HOL [NPW02]. The correctness is based on the model VAMP assembly with devices from Section 3.1.4 and the reordering theory developed in Section 3.2. The driver writes a $4K$ page (8 sectors) from the processor's memory, starting at address ma, to the disk, starting at sector sa. Its code is shown in Fig. 4.2. Arrows indicate jump targets. Note, that according to the delayed PC, instructions in delay slots are always executed.

The code is parametrized over: (i) The index id_{hd} of the hard disk, and (ii) the memory location of the two call parameters ma and sa. The code is later on embedded into a C0 function call. Thus, the call parameters are expected to be stored either as function arguments or local variables. The two code parameters sa_{off} and ma_{off} denote the offsets in bytes of the two call parameters in the top function frame. Remember, that the address of the latter is stored in register topl.

Formally, we denote the code by the function $d\text{-}code$ which takes the code parameters and return the implementation of the driver depicted in Figure 4.2:

$$d\text{-}code :: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to instr_T \ list$$

We denote with d-length the length of the driver code in words:

$$\textsf{d-length} = length(d\text{-}code(id_{\text{hd}}, sa_{\text{off}}, ma_{\text{off}}))$$

The code can be structured into five main parts. In part 0, we first load the two parameters, ma and sa of the driver call to the registers 2 and 16 respectively. After loading the parameters, the start sector index sa is decomposed into the sector number in step (0.3), cylinder low in steps (0.4) and (0.5), cylinder high in steps (0.6) and (0.7), and drive index in steps (0.8) to (0.10). Finally, the values for the write command, the interrupt disable flag and the sector count are written into registers in steps (0.11) to (0.13).

In part 1, first, the disk address is saved to register 1. Then the command parameters are written to the disk's configuration ports. The interrupt mode is disabled in step (1.2), the count of sectors to process is stored in step (1.3), the sector address is stored in steps (1.4) to (1.7) and, finally, the write command is issued in step (1.8).

Each iteration of the outer loop in steps (2.1) to (5.3) copies one sector from the main memory of the processor to the sector memory of the disk. One sector consists of 128 words. The first inner loop copies word after word first from the processor's memory to register 3 and subsequently to the internal disk buffer. When the buffer is full (i.e., after one complete sector) the driver enters the second inner loop, which polls on the status register until the hard disk has written its buffer.

4.2.1 Correctness Theorem

The correctness of the driver is proven for all valid execution sequences. In a nutshell, it states that the driver execution finally terminates, and that then the page located in the processor memory is copied to the sector memory of the disk.

Preconditions of the theorem Note, that in the following, the code parameters sa_{off} and ma_{off} are not passed explicitly to all predicates as parameters.

- *Assumptions on the environment.* We (can) prove the driver correctness only for those execution sequences, which are well-typed and in which the processor and the trigger of the hard disk are live, i.e. given a sequence *seq*, we require:

$$\begin{aligned}&\mathit{well\text{-}typed}(\mathit{seq})\\ \wedge\ &\mathit{hd\text{-}trigger\text{-}live}(\mathit{seq})\\ \wedge\ &\mathit{proc\text{-}live}(\mathit{seq})\end{aligned}$$

  These conditions are subsumed in the set of valid sequences Seq_V defined in Section 4.1.

- *Assumptions on the hard disk.* We assume that the index of the hard disk is in range, and that the hard disk connected to the processor is in initial configuration (i.e. the control state is idle, the buffer pointer is zero and the configuration is valid):

$$\mathit{driver\text{-}precond}_{\text{hd}} :: (\mathfrak{D} \to S_{\mathfrak{D}}) \to \mathfrak{D} \to \mathit{bool}$$

$$\begin{aligned}&\mathit{driver\text{-}precond}_{\text{hd}}(\mathit{devs}, \mathit{id}_{\text{hd}}) \equiv\\ &\qquad \mathit{id}_{\text{hd}} \in \mathfrak{D}\\ &\quad \wedge\ \ \mathit{initial\text{-}disk}(\mathit{devs}(\mathit{id}_{\text{hd}}))\end{aligned}$$

- *Assumptions on the assembly machine.* We assume that the assembly configuration is in a valid state, in which the system mode is set and all interrupts are disabled. These conditions are subsumed by the precondition of the ISA-assembly transfer theorem.

  Moreover, we require that the driver code is correctly stored in the assembly machine, i.e. it is within regular memory and it is pointed to by the current delayed program counter.

  Thus, we get for the assembly machine the following precondition:

$$\textit{driver-precond}_{\text{proc}} :: C_{asm} \rightarrow \textit{bool}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\textit{driver-precond}_{\text{proc}}(\textit{asm}) \equiv} \\ & \textit{valid-asm}(\textit{asm}) \\ \wedge & \textit{asm}.\mathsf{spr!MODE} = 0 \\ \wedge & \textit{asm}.\mathsf{spr!SR} = 0 \\ \wedge & \textit{asm}.\mathsf{dpc} + \mathsf{d\text{-}length} < \mathsf{device\text{-}border} \\ \wedge & \textit{to-instr-list}(\textit{asm}.\mathsf{mm}, \textit{asm}.\mathsf{dpc}, \mathsf{d\text{-}length}) = \textit{d-code}(\textit{id}_{\text{hd}}, \textit{sa}_{\text{off}}, \textit{ma}_{\text{off}}) \end{array}$$

- *Assumptions on call parameters* For a correct reading of the call parameters, we have to ensure that the corresponding memory locations are in range, aligned and do not intersect with the device domain.

  Moreover, we require that all memory addresses from which the driver reads are word aligned and do not overlap with device domains, i.e. the start address ma from which the driver reads and the last address $ma + 128 \cdot 8$ are valid.

  For the sector addresses to which the driver writes, we assume that they are within the maximum sector count of the hard disk.

$$\textit{driver-precond}_{\text{args}} :: C_{asm} \times \mathbb{N} \times \mathbb{N} \rightarrow \textit{bool}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\textit{driver-precond}_{\text{args}}(\textit{asm}, \textit{sa}, \textit{ma}) =} \\ & \textit{valid-ad}(\textit{i2n}(\textit{asm}.\mathsf{gpr!topIm}) + \textit{ma}_{\text{off}}) \\ \wedge & \textit{valid-ad}(\textit{i2n}(\textit{asm}.\mathsf{gpr!topIm}) + \textit{sa}_{\text{off}}) \\ \wedge & \textit{ma} = \textit{i2n}(\textit{asm}.\mathsf{mm}((\textit{i2n}(\textit{asm}.\mathsf{gpr!topIm}) + \textit{ma}_{\text{off}}) \div 4)) \\ \wedge & \textit{sa} = \textit{i2n}(\textit{asm}.\mathsf{mm}((\textit{i2n}(\textit{asm}.\mathsf{gpr!topIm}) + \textit{sa}_{\text{off}}) \div 4)) \\ \wedge & \textit{valid-ad}(\textit{ma}) \\ \wedge & \textit{valid-ad}(\textit{ma} + 128 \cdot 8 \cdot 4) \\ \wedge & \textit{sa} + 8 < S \end{array}$$

Postconditions of the theorem

- *Hard disk.* The main result of the theorem is that finally the data is copied from the processor's memory to the sector memory of the hard disk. Moreover, we have to show that the hard disk is again in initial configuration (i.e. it is valid, the control state is idle and the buffer pointer is zero).

$$\begin{aligned}
&\mathit{driver\text{-}post}_{\mathrm{hd}}(\mathit{asmd}', \mathit{asmd}, \mathit{id}_{\mathrm{hd}}, \mathit{sa}, \mathit{ma}) \equiv \\
&\quad \mathit{initial\text{-}disk}(\mathit{asmd}'.\mathsf{devs}(\mathit{id}_{\mathrm{hd}})) \\
\wedge &\quad \mathit{asmd}'.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{sm} = \\
&\quad (\mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}})).\mathsf{sm}_{8\cdot 128}(128 \cdot \mathit{sa}) := \mathit{asmd}.\mathsf{mm}(\mathit{ma} \div 4)
\end{aligned}$$

Note, that this predicate also ensures that sectors other than the ones specified by the call parameter have not been changed by the driver.

- *Processor.* For the processor we show that neither the main memory nor the special purpose registers were altered. For a correct embedding of inline assembly code (cf. Section 3.4), we also have to ensure that registers used by the compiler have not been changed.

  Moreover, we show that the program counters point at the first instruction after the driver code. For the driver code we also claim, that it has not been manipulated.

$$\begin{aligned}
&\mathit{driver\text{-}post}_{\mathrm{proc}}(\mathit{asm}, \mathit{asm}') \equiv \\
&\quad \mathit{asm}'.\mathsf{mm} = \mathit{asm}.\mathsf{mm} \\
\wedge &\quad \mathit{asm}'.\mathsf{spr} = \mathit{asm}.\mathsf{spr} \\
\wedge &\quad \mathit{asm}'.\mathsf{gpr!sbase} = \mathit{asm}.\mathsf{gpr!sbase} \\
\wedge &\quad \mathit{asm}'.\mathsf{gpr!toph} = \mathit{asm}.\mathsf{gpr!toph} \\
\wedge &\quad \mathit{asm}'.\mathsf{gpr!toplm} = \mathit{asm}.\mathsf{gpr!toplm} \\
\wedge &\quad \mathit{asm}'.\mathsf{dpc} = \mathit{asm}.\mathsf{dpc} + \mathsf{d\text{-}length} \\
\wedge &\quad \mathit{to\text{-}instr\text{-}list}(\mathit{asm}.\mathsf{mm}, \mathit{asm}.\mathsf{dpc}, \mathsf{d\text{-}length}) = \mathit{d\text{-}code}(\mathit{id}_{\mathrm{hd}}, \mathit{sa}_{\mathrm{off}}, \mathit{ma}_{\mathrm{off}})
\end{aligned}$$

- *ISA-assembly translation.* The pervasive approach in Verisoft requires that all results can be finally transferred down to the gate-level implementation of the hardware. Since assembly is only an intermediate model, we have to ensure that driver correctness can also be expressed at the level of ISA. Thus, the precondition $\mathit{isa\text{-}asm\text{-}precond}_{\mathrm{dyn}}$ of the simulation Theorem 1 has to be guaranteed (requires the predicate *step-properties* to hold in each step).

- *Reordering.* Remember that the reordering theory described in Section 3.2 was mainly developed for pure computations, i.e. computations during which the processor controls at most one device. Thus, to apply the reordering results on the hard disk driver, we have to prove that during execution only the hard disk is accessed:

$$\mathit{only\text{-}hd\text{-}access} :: C_{\mathrm{asmd}} \times \mathit{Seq}_T \times \mathbb{N} \rightarrow \mathit{bool}$$

$$\mathit{only\text{-}hd\text{-}access}(\mathit{asmd}, \mathit{seq}, N) \equiv \mathit{pure}(\mathit{asmd}, \mathit{seq}, N, \mathit{id}_{\mathrm{hd}})$$

Stating the Theorem The assembly driver correctness theorem claims for each valid execution sequence the existence of a step number N after which the driver has finished execution and has written a $4K$ page from the processor's memory, starting at address ma, to the disk, starting at sector sa.

Theorem 10 (Assembly driver correctness)

$$\begin{array}{l}
\forall asmd_0, id_{hd}, sa, ma.\,(\\
\quad\quad driver\text{-}precond_{hd}(asmd_0.\mathsf{devs}, id_{hd}) \\
\quad \wedge \;\; driver\text{-}precond_{proc}(asmd_0.\mathsf{proc}) \\
\quad \wedge \;\; driver\text{-}precond_{args}(asmd_0.\mathsf{proc}, sa, ma) \\
\quad \Longrightarrow \\
\quad \forall seq \in Seq_V.\,(\\
\quad\quad \exists asmd', N. \\
\quad\quad\quad\quad asmd' = \varDelta_{asmd}(asmd_0, seq, N) \\
\quad\quad\quad \wedge \;\; driver\text{-}post_{hd}(asmd_0, asmd', id_{hd}, sa, ma) \\
\quad\quad\quad \wedge \;\; driver\text{-}post_{proc}(asmd_0.\mathsf{proc}, asmd'.\mathsf{proc}) \\
\quad\quad\quad \wedge \;\; isa\text{-}asm\text{-}precond_{dyn}(asmd_0, asmd_0.\mathsf{proc.dpc}, \mathsf{d\text{-}length}, seq, N) \\
\quad\quad\quad \wedge \;\; only\text{-}hd\text{-}access(asmd_0, seq, N)))
\end{array}$$

4.2.2 Proving the theorem

Proof Methodology. We apply the reordering theory developed in Section 3.2 to sequentialize reasoning about the concurrent system as far as possible. The device under control is the hard disk, with index $id_{\text{hd}} \in \mathfrak{D}$; other devices are not accessed. Stability holds for the hard disk in all states, except for those in which a sector is transfered between the buffer and the sector memory.

Lemma 25 (Stability of hard disk)

$$(c_{hd}.\mathsf{cs} \neq \mathsf{bwr} \wedge c_{hd}.\mathsf{cs} \neq \mathsf{brd}) \implies stable(c_{hd})$$

For parts of the code in which either no device at all or only a stable hard disk is accessed, correctness can be proven by reasoning solely on steps of the assembly machine and by ignoring external device steps.

We prove properties on the assembly code by forward-style reasoning. No logic or automatic inference system is applied. Rather, we annotate each line i of the code with a predicate $Q_i :: C_{\text{asmd}} \rightarrow bool$ over the combined system. We require Q_i to hold whenever the assembly machine is at instruction i of the hard disk driver. The latter condition is denoted by the following predicate:

$$@^i :: C_{asm} \rightarrow bool$$

$$\begin{array}{rcl}
@^i(asm) \equiv & & asm.\mathsf{dpc} = asmd^0.\mathsf{proc.dpc} +_{32} 4 \cdot i \\
& \wedge & asm.\mathsf{pc} = asmd^0.\mathsf{proc.dpc}
\end{array}$$

Formally, we prove for each line i of the device driver:

$$@^i(asm) \implies Q_i(asm)$$

Note, that a program logic would enable us to generate these line properties automatically (except for loop invariants).

Moreover, we use a technique introduced by Leinenbach [Lei08] to simplify reasoning about general invariants which have to hold in each step of the assembly computation (see blow).

Basic safety property. Some properties have to hold in each step of the driver execution, we call those *basic invariants*. Basic invariants split in:

- General proof obligations, i.e. properties which are not driver-specific. These are (i) validity of the assembly state, and (ii) dynamic preconditions of the ISA-assembly translation (given by *step-properties* defined in Secton 3.1.5).
- Driver-specific conditions. During execution we have to ensure that the code, certain registers and the memory of the assembly machine have not been manipulated. We group these conditions by a predicate $\mathit{driver\text{-}post}'_{\mathrm{hd}}$, which is equal to its unprimed counterpart, except that the second last conjunct (on the delayed program counter) is omitted.

Suppose the initial configuration $asmd_0$ is given as parameter, then the basic invariant of the driver code is defined by:

$$\mathit{basic\text{-}inv} :: C_{asm} \rightarrow \mathit{bool}$$

$$\begin{aligned} &\mathit{basic\text{-}inv}(asm) \equiv \\ &\qquad \mathit{valid\text{-}asm}(asm) \\ &\wedge \quad \mathit{step\text{-}properties}(asm, asmd^0.\mathsf{proc.dpc}, \mathsf{d\text{-}length}) \\ &\wedge \quad \mathit{driver\text{-}post}'(asmd^0.\mathsf{proc}, asm) \end{aligned}$$

Discharging the general proof obligations *validity* and *step-properties* in each step by hand, can be extremely cumbersome. Even though most instructions, do not interfere with them, still, in the mechanic proof system, they have to be carried to each assumption and conclusion, unpacked and handled. This could lead to a significant proof overhead, a much higher complexity, and slow down the prover considerably.

The solution is to use an extended assembly semantics, which by definition ensures the maintenance of these invariants. This semantics is defined via a set of lemmas, one for each instruction type. For example, for addition, instead of using the plain assembly semantics, we resort to the following lemma:

Lemma 26 (Addition)

$$
\begin{array}{ll}
& \textit{step-properties}(asm, \textit{start-adr}, \textit{prog-len}) \\
\wedge & \textit{valid-asm}(asm) \\
\wedge & \textit{current-instr}(asm) = \textit{Iaddi dr sr i} \\
\Longrightarrow & \\
& \delta_{asm}(asm) = \quad asm[\ \mathsf{dpc} := asm.\mathsf{dpc} +_{32} 4, \\
& \qquad\qquad\qquad\qquad \mathsf{pc} := asm.\mathsf{pc} +_{32} 4, \\
& \qquad\qquad\qquad\qquad \mathsf{gpr} := asm.\mathsf{gpr}[dr := asm.\mathsf{gpr}!(sr) +_{32} i]] \\
\wedge & \textit{step-properties}(\delta_{asm}(asm), \textit{start-adr}, \textit{prog-len}) \\
\wedge & \textit{valid-asm}(\delta_{asm}(asm))
\end{array}
$$

Initialization. The initialization consists of two main chunks: in part 0, addresses, commands and configuration values are computed and written to the processor's registers. Most importantly, the start sector address *sa* is decomposed into the different *lba* blocks. For part 0, we prove that any computation starting at step (0.1) and with $Q_{0.1}$ as the preconditions of the theorem, finally will lead to step (1.1) where the line predicate $Q_{1.1}$ is fulfilled and the basic invariant holds in each step in between. During its execution, the processor does not access the hard disk at all. Thus, it suffices to prove correctness (and termination) in the assembly model without devices. We apply Lemma 5 to generalize this result to an arbitrary execution of the interleaved system, where the free variable Q of the lemma is instantiated to the basic invariant. Note, that the precondition of the lemma — namely that no device is accessed — is also discharged on the level of pure assembly.

Then, in part 1, the configuration ports of the hard disk are set. We prove that any computation starting at step (1.1) and with $Q_{1.1}$ fulfilled, finally will lead to step (1.8) where the the first instantiation of the outer loop invariant, defined in the next paragraph, holds: $\textit{outer-loop-inv}(asmd, 0)$. During its execution, the processor only accesses the hard disk. Moreover, the hard disk remains in the control state idle which is according to Lemma 25 stable. Thus, it suffices to prove correctness (and termination) for the sequential assembly model with devices, in which only the processor takes steps (i.e. for the empty execution sequence defined in Section 3.2.2). We apply Lemma 6 to generalize this result to an arbitrary execution of the interleaved system. The free variable Q of the lemma is again instantiated to the basic invariant. Note, that the precondition of the lemma — only hard disk access and the stability of the hard disk — are also discharged on the level of sequential assembly.

The initialization part, decomposes the sector start address and writes each component to the corresponding component of the hard disk:

$$
\begin{array}{ll}
& asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{lba.snumb} = sa \wedge_{32} 256, \\
\wedge & asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{lba.cyll} = (sa >>^{l}_{32} 8) \wedge_{32} 256, \\
\wedge & asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{lba.cylh} = (sa >>^{l}_{32} 16) \wedge_{32} 256, \\
\wedge & asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{lba.drivehead} = (((sa >>^{l}_{32} 24) \wedge_{32}) +_{32} 224) \quad \bmod 16
\end{array}
$$

We have to prove correctness of this decomposition, i.e. the following condition of the outer loop invariant has to hold:

$$\mathit{lba\text{-}to\text{-}nat}(\mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{lba}) = \mathit{sa}$$

First, we show that bit-level shift can be interpreted as division, and bit-level AND with sequences of ones as modulo on naturals:

Lemma 27 (Shift as modulo/ And as division)

$$\begin{array}{ll} & i > 0 \land z > 0 \land sa \geq 0 \\ \land & (sa >>^{l}_{32} i) \land_{32} (2^z - 1) = (sa \div 2^i) \mod 2^z \end{array}$$

After applying this lemma on our goal, we only have to prove the following transformation to be correct:

Lemma 28 (Correctness of lba computation)

$$\begin{array}{lll} & sa < 2^{28} & \\ \Longrightarrow & \mathit{lba\text{-}to\text{-}nat} & (\mathsf{snumb} = sa \mod 256, \\ & & \mathsf{cyll} = (sa \div 256) \mod 256, \\ & & \mathsf{cylh} = (sa \div 256^2) \mod 256, \\ & & \mathsf{drivehead} = ((sa \div 256^3) \mod 16) + 224 \mod 16) \\ & = sa & \end{array}$$

This lemma is proven by straightforward arithmetic.

Loop invariants The *outer loop*, parts (2) to (5), is traversed 8 times, writing a sector to the disk in each iteration. Let i denote the number of sectors so far processed. For the hard disk the outer loop invariant maintains that: (i) interrupts are disabled, (ii) the sector address has been correctly decomposed and stored in the *lba* component of the disk, (iii) the component scnt hold the number of sectors still to process, i.e. $8-i$, (iv) the buffer pointer is zero, (v) i sectors have already been copied correctly from the processor's memory to the sector memory, (vi) if not all sectors have been already copied, the hard disk is in state pwr otherwise in state idle.

For the processor the outer loop invariant maintains that: (i) the disk address is correctly stored in register 1, (ii) the memory address of the next sector to copy is stored in register 2, (iii) the number of sectors still to process is stored in register 12, (iv) the program control is either at the beginning of the loop ($i < 8$), or at the first instruction after loop ($i = 8$).

Formally, the invariant on the complete system reads as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{outer\text{-}loop\text{-}inv}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i) =} \\
 & \mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{bp} = 0 \\
\wedge & \mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{ien} = \mathsf{false} \\
\wedge & \mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{scnt} = 8 - i \\
\wedge & \mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{cs} = \textbf{if } i < 8 \textbf{ then } \mathsf{pwr} \textbf{ else } \mathsf{idle} \\
\wedge & \mathit{lba\text{-}to\text{-}nat}(\mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{lba}) = \mathit{sa} + i \\
\wedge & \mathit{asmd}.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{sm} = \\
 & \quad (\mathit{asmd}_0.\mathsf{devs}(\mathit{id}_{\mathrm{hd}}).\mathsf{sm}_{128 \cdot i}(\mathit{sa}) := \mathit{asmd}_0.\mathsf{proc.mm}(\mathit{ma} \div 4)) \\
\wedge & \mathit{asmd}.\mathsf{proc.gpr!}(1) = D0(\mathit{id}_{\mathrm{hd}}) \\
\wedge & \mathit{asmd}.\mathsf{proc.gpr!}(2) = \mathit{ma} + 4 \cdot 128 \cdot i \\
\wedge & \mathit{asmd}.\mathsf{proc.gpr!}(12) = 8 - i \\
\wedge & (\textbf{if } i < 8 \textbf{ then } @^{2.1} \textbf{ else } @^{(5.3+1)})(\mathit{asmd}.\mathsf{proc})
\end{array}$$

The correctness of the driver follows from proving that the invariant is indeed maintained by the outer loop:

Lemma 29 (Correctness of outer loop)

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall \mathit{seq} \in \mathit{Seq}_V \,.} \\
 & i < 8 \wedge \mathit{outer\text{-}loop\text{-}inv}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i) \\
\Longrightarrow & (\exists T \,.\, \mathit{outer\text{-}loop\text{-}inv}(\mathit{asmd}^{\mathit{seq},T}, \mathit{sa}, \mathit{ma}, i+1))
\end{array}$$

For the proof of this lemma for $j > 0$ we have to reason about the data copy and the polling loop, parts (3) and (4).

The data copy loop writes a single sector from the memory to the disk buffer. We define the invariant $\mathit{inner\text{-}loop\text{-}inv}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i, k)$ in which the inner loop has been traversed k times as part of traversal i of the outer loop. Again, invariants on control, counters, and device state are maintained. Most importantly the invariant maintains that (i) the buffer pointer is set to k, (ii) the hard disk is in state pwr if there are still words to be copied, otherwise it is in state bwr, (iii) k words have been copied from the memory to the buffer, (iv) the memory address of the next word to copy is stored in register 2, (v) the number of words still to process is stored in register 10, (vi) the program control is either at the beginning of the loop ($k < 128$), or at the first instruction after loop ($k = 128$).

Formally, we have:

$$\begin{aligned}
&\mathit{inner\text{-}loop\text{-}inv}(asmd, sa, ma, i, k) = \\
&\quad asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{ien} = \mathsf{false} \\
&\wedge\ \mathit{lba\text{-}in\text{-}range}(asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{lba}) \\
&\wedge\ asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{scnt} = 8 - i \\
&\wedge\ \mathit{lba\text{-}to\text{-}nat}(asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{lba}) = sa + i \\
&\wedge\ asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{sm} = \\
&\qquad (asmd_0.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{sm}_{128 \cdot i}(sa) := asmd_0.\mathsf{proc.mm}(ma \div 4)) \\
&\wedge\ asmd.\mathsf{proc.gpr!}(1) = D0(id_{\mathrm{hd}}) \\
&\wedge\ asmd.\mathsf{proc.gpr!}(12) = 8 - i \\
&\wedge\ asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{cs} = \textbf{if } k < 128 \textbf{ then } \mathsf{pwr} \textbf{ else } \mathsf{bwr} \\
&\wedge\ asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{bp} = k \mod 128 \\
&\wedge\ asmd.\mathsf{proc.gpr!}(2) = ma + 4 \cdot 128 \cdot i + k \\
&\wedge\ asmd.\mathsf{proc.gpr!}(10) = 128 - k \\
&\wedge\ asmd.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{buf} = \\
&\qquad (asmd_0.\mathsf{devs}(id_{\mathrm{hd}}).\mathsf{buf}_k(0) := asmd_0.\mathsf{proc.mm}((ma \div 4) + 128 \cdot i)) \\
&\wedge\ (\textbf{if } k < 128 \textbf{ then } @^{3.1} \textbf{ else } @^{4.1})(asmd.\mathsf{proc})
\end{aligned}$$

We prove, that the inner loop maintains the invariant defined above:

Lemma 30 (Correctness of first inner loop)

$$\begin{aligned}
&\forall seq \in Seq_V . \\
&\qquad\quad k < 128 \wedge i < 8 \\
&\quad \wedge \quad \mathit{inner\text{-}loop\text{-}inv}(asmd, sa, ma, i, k) \\
&\quad \Longrightarrow \quad (\exists T . \mathit{inner\text{-}loop\text{-}inv}(asmd^{seq,T}, sa, ma, i, k+1))
\end{aligned}$$

During the inner loop, the processor only accesses the hard disk. Moreover, the hard disk remains in the control state pwr which is according to Lemma 25 stable. Thus, it suffices to prove correctness (and termination) for the sequential assembly model with devices, in which only the processor takes steps (i.e. for the empty execution sequence). We apply Lemma 6 to generalize this result to an arbitrary execution of the interleaved system.

Polling So far, code correctness could be proven by applying sequential assembly and hard disk semantics. This was possible, since the processor either accessed no device at all, or the accessed hard disk was stable. That does not hold anymore at the end of the first inner loop, where the disk may be in the the non-stable state bwr (cf. Figure 4.3).

First, we split the inner loop invariant into the two parts $\mathit{inner\text{-}loop\text{-}inv}_{\mathrm{hd}}$, grouping all assumptions on the hard disk, and $\mathit{inner\text{-}loop\text{-}inv}_{\mathrm{proc}}$, grouping all assumptions on the processor, except for the last conjunct on the program control. Moreover, the predicate *polling* indicates that the processor is within the polling loop:

$$\mathit{polling}(asm) = @^{4.1}(asm) \vee @^{4.2}(asm) \vee @^{4.3}(asm) \vee @^{4.4}(asm)$$

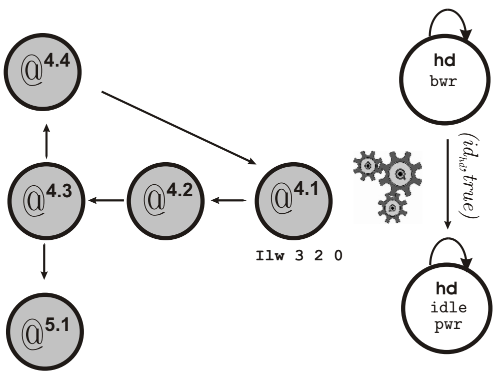


Figure 4.3: During the polling loop

Since, valid sequences ensure liveness of the hard disk trigger, we deduce that finally a step is reached after which the trigger is scheduled for the first time:

Lemma 31 (Smallest element)

$$\begin{aligned}
&\forall seq \in Seq_V, T\,.\\
&\quad(\exists T'.\\
&\qquad seq(T') = (id_{hd}, \mathsf{true})\\
&\quad\wedge\ \ (\forall T < T'' < T'.seq(T'') \neq (id_{hd}, \mathsf{true})))
\end{aligned}$$

Within this time, the processor will neither leave the polling loop nor invalidate the invariants:

Lemma 32 (During Polling)

$$\begin{aligned}
& && inner\text{-}loop\text{-}inv(asmd, sa, ma, i, 128)\\
&\wedge && @^{4.1}(asmd.\mathsf{proc})\\
&\wedge && (\forall t \leq T\,.\,seq(t) \neq (id_{D\,hd}, true))\\
&\Longrightarrow && (\forall t \leq T\,.\,polling(asmd^{seq,T}.\mathsf{proc})\\
&\wedge && inner\text{-}loop\text{-}inv_{hd}(asmd^{seq,T}.\mathsf{proc}, sa, ma, i, 128)\\
&\wedge && inner\text{-}loop\text{-}inv_{proc}(asmd^{seq,T}.\mathsf{proc}, sa, ma, i, 128))
\end{aligned}$$

For the proof, all device steps except for the hard disk can be ignored (they are not accessed by the processor). Moreover the hard disk will not change its state, because no trigger is scheduled and reading the status register does not alter the disk.

Finally, the trigger is true, and in the following step the disk has transferred the buffer to the sector memory and switches to a stable state again, i.e. $inner\text{-}loop\text{-}inv_{\mathrm{hd}}(asmd^{seq,T}.\mathsf{proc}, sa, ma, i+1, 0)$ holds.

We prove, that once the hard disk was triggered, the polling loop will finally again reach step (4.1) where the status register is read:

Lemma 33 (End of Polling)

$$\begin{array}{ll} & \mathit{inner\text{-}loop\text{-}inv}_{\mathit{proc}}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i, 128) \\ \wedge & \mathit{polling}(\mathit{asmd}.\mathsf{proc}) \\ \wedge & \mathit{inner\text{-}loop\text{-}inv}_{\mathit{hd}}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i+1, 0) \\ \Longrightarrow & (\exists T\,.\, @^{4.1}(\mathit{asmd}.\mathsf{proc}) \\ \wedge & \mathit{inner\text{-}loop\text{-}inv}_{\mathit{hd}}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i+1, 0) \\ \wedge & \mathit{inner\text{-}loop\text{-}inv}_{\mathit{proc}}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i, 128)) \end{array}$$

Once the processor is at step (4.1) and the disk is stable, the polling loop will be finally left and the next outer loop invariant will be established:

Lemma 34 (End of Polling 2)

$$\begin{array}{ll} & \mathit{inner\text{-}loop\text{-}inv}_{\mathit{proc}}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i, 128) \\ \wedge & @^{4.1}(\mathit{asmd}.\mathsf{proc}) \\ \wedge & \mathit{inner\text{-}loop\text{-}inv}_{\mathit{hd}}(\mathit{asmd}, \mathit{sa}, \mathit{ma}, i+1, 0) \\ \Longrightarrow & (\exists T\,.\, \mathit{outer\text{-}loop\text{-}inv}(\mathit{asmd}^{\mathit{seq},T}, \mathit{sa}, \mathit{ma}, i+1, 0)) \end{array}$$

The last two lemmas are proven sequentially for empty execution sequences. Since the disk is stable we can generalize these results to arbitrary interleaved executions by applying Lemma 25.

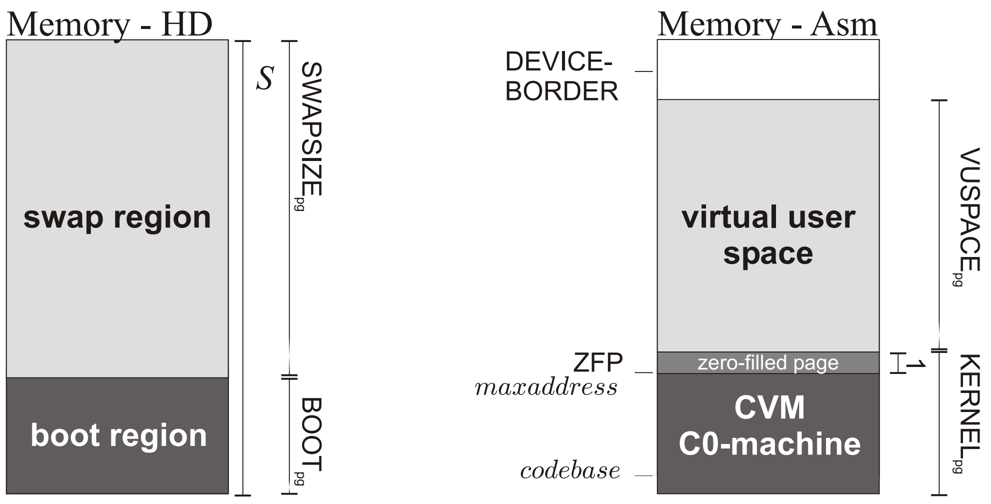


Figure 4.4: Memory layout of SWAP/CVM on hard disk/target machine

4.3 C0 Driver

In this section we embed the assembly hard disk driver (cf. Section 4.2) into a C0 call, specify its effects in terms of XCall semantics (cf. Section 3.5) and prove its correctness.

With this final theorem, we discharge the implementation correctness proof obligation of the XCalls used in the context of the page-fault handler verification. Moreover, page-fault handler correctness itself gets transferred by this theorem from C0 with XCalls down to VAMP assembly with devices.

4.3.1 Notation and constants

The memory of the target machine is organized in pages. A size of a page in words is denoted by PAGE-SIZE $= 1024$. We denote the page index of some memory address ad by $px(ad) \equiv ad \div 2^{12}$.

The driver is called as part of the page-fault handler, which in turn is invoked by the kernel. Hence, the C0 machine of the kernel is the top-most reference model for the following verification of the driver. In Figure 4.4 the memory layout of the machine is depicted. Basically, the memory of the target machine splits into two parts: the kernel memory, and the virtual memory space of the user machines. The former allocates KERNEL$_{pg}$ many pages. It consists not only of the C0 machine implementing the kernel, but also of a special page, called zero-filled paged with page index ZFP $=$ KERNEL$_{pg} - 1$ (i.e. last page of the kernel memory; used for lazy allocation of new pages). The size in pages of the virtual user space space is denoted by the constant

$\mathsf{VUSPACE}_{\mathrm{pg}}$.

The sector memory of the hard disk is logically divided into two parts: a boot region and the rest, which is used by the page-fault handler to swap out pages. The boot region is never accessed by the page-fault handler. Its size is denoted by the constant $\mathsf{BOOT}_{\mathrm{pg}}$. The constant $\mathsf{SWAPSIZE}_{\mathrm{pg}}$ denotes the size of the hard disk memory in pages without the pages used for booting, i.e. $\mathsf{SWAPSIZE}_{\mathrm{pg}} = S \div 8 - \mathsf{BOOT}_{\mathrm{pg}}$.

4.3.2 Embedding the driver in C0

The driver presented here is used in the generic Verisoft kernel CVM to perform page swap-out. This is achieved by embedding the assembly code into the two C0 functions with names $\mathsf{writePage}_{\mathrm{i}}$ and $\mathsf{readPage}_{\mathrm{i}}$.

We describe the implementation of the C0 driver for writing pages by the following function declaration:

$$\mathsf{writePage}_{\mathrm{idecl}} :: \mathit{fun\text{-}decl}_T$$

It is made up of the following four fields:

- **Parameters.** Like the assembly driver, it gets two parameters: 1. The start address $\mathsf{ma} \in \mathit{var}_n$ of the page to write, and 2. the destination sector address $\mathsf{sa} \in \mathit{var}_n$.

- **Local variables** No local variables are required.

- **Return type** A call to the hard disk driver returns always with the boolean value true. The implementation does not treat incorrect calls. Conceivable failures may be due to lacking of swap memory or invalid addresses. Not-treating these sources of possible failures both in specification and even in the implementation follows a simple rule: For non top-level functionalities we only specify *correct* behavior. Non top-level includes all functions which are only called in verified code, i.e. which is not delivered to some end user.

  Though in case of our hard disk driver it gets the obligation of the calling program, i.e. the page fault handler to ensure that parameters are always valid and the memory size is not exceeded.

- **Program body** The program body consists of the inline assembly code and a return statement. Remember, that the assembly driver is parametrized over the offsets of the call parameters in the current stack frame and the device id. The corresponding paramter offsets sa_{off} and ma_{off} of the C0 function call are 16 and 20 (the first four words are used to store the function header, cf. Section 3.3). We assume that the hard disk is connected as device with id zero.

Formally, we describe the function declaration by:

$$\begin{aligned}&\mathit{writePage}_{\text{idexl}} \equiv\\&\quad(\mathsf{params} = [(\mathsf{ma}, \mathsf{Int}), (\mathsf{sa}, \mathsf{Int})],\\&\quad\;\mathsf{locals} = [],\\&\quad\;\mathsf{ret\text{-}type} = \mathsf{Boolean},\\&\quad\;\mathsf{body} = \mathsf{Asm}(\mathit{d\text{-}code}(0, 16, 20);; \mathsf{Return}(\mathsf{Lit}(\mathsf{BoolV}(\mathsf{true}))))\end{aligned}$$

The read case of the driver is implemented analogously by the function declaration $\mathsf{readPage}_{\text{idecl}} \in \mathit{fun\text{-}decl}_T$.

4.3.3 Specifying the driver

We specify the driver by defining (i) the extended state, (ii) the corresponding abstraction relation, and (iii) the semantics of the XCalls for reading and writing pages.

Extended State The extended state is an abstraction of the non-system memory of the machine running the driver (i.e. the page fault handler which is ran by the kernel) and the hard disk connected to that machine. The state is used to specify the effects of read and write operations to and from the hard disk. It has record type pfhX_T with the following two components, which are both modeled as mappings from page indices to page contents, where the latter consist of word-addressable memories:

$\mathsf{mem} :: \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$	User part of the physical memory, which is not reachable by C0.
$\mathsf{swap} :: \mathbb{N} \rightarrow \mathbb{N}$	Content of the hard disk's sector memory excluding the boot region.

Abstraction relation According to the XCall theory developed in Section 3.5, we have to provide the abstraction relation $\mathit{xconsis}_{\text{ex}}$. It consists of the following two parts:

- The abstraction relation $\mathit{xconsis}_{\text{swap}}$ maps the extended swap component to the sector memory of the hard disk, except for the boot sectors. Moreover, we require that the hard disk is in initial state. Formally, we get:

$$\begin{aligned}&\mathit{xconsis}_{\text{swap}}(c_{hd}, \mathit{xswap}) \equiv\\&\qquad \mathit{initial\text{-}disk}(c_{hd})\\&\;\wedge\quad (\forall k < \mathsf{SWAPSIZE}_{\text{pg}}, i < \mathsf{PAGE\text{-}SIZE}\,.\\&\qquad\quad c_{hd}.\mathsf{sm}(\mathsf{PAGE\text{-}SIZE} \cdot (\mathsf{BOOT}_{\text{pg}} + k) + i) = \mathit{xswap}(k)(i))\end{aligned}$$

- The abstraction relation $xconsis_{\text{swap}}$ maps the extended memory component to the parts of the processor's main memory representing the user virtual memory space.

  Only pages which lay outside the range of the C0 machine, which implements the kernel are covered by the abstraction. The kernel memory contains not only the C0 machine implementing it, but also the zero-filled page. Thus, all pages, starting at the zero-filled page and ending at the last page belonging to the virtual user space are contained in the extended memory component

$$\begin{aligned}&xconsis_{\text{mem}}(mm, xmem) \equiv\\ &\quad \forall k < (\mathsf{ZFP} + \mathsf{VUSPACE}_{\text{pg}})\,,\, i < \mathsf{PAGE\text{-}SIZE}\,.\\ &\qquad k \geq \mathsf{ZFP} \implies\\ &\qquad i2n(mm(\mathsf{PAGE\text{-}SIZE} \cdot k + i)) = xmem(k - \mathsf{ZFP})(i)\end{aligned}$$

We get for the overall abstraction relation on the extended state:

$$\begin{aligned}xconsis_{\text{driver}}(pfhX, (mm, c_{hd})) = &\quad xconsis_{\text{mem}}(mm, pfhX.\mathsf{mem})\\ \wedge &\quad xconsis_{\text{swap}}(c_{hd}, pfhX.\mathsf{swap})\end{aligned}$$

XCall semantics Before introducing the XCall semantics, we define two functions, which map page indices of the main memory and the sector memory in the implementation to their counterparts in the extended state.

For memory addresses we skip the kernel pages, except for the zero-filled page: $adjust_{\text{mem}}(ad) \equiv px(ad - 4 \cdot \mathsf{ZFP} \cdot \mathsf{PAGE\text{-}SIZE})$. For sector addresses we skip the boot pages: $adjust_{\text{swap}}(ad) \equiv (ad \div 8) - \text{BOOT}_{\text{pg}}$.

As mentioned before it suffices only to verify behavior of the driver for the non-failure case. Therefore, we identify a set of conditions over the parameters of the extended semantics, for which the XCall semantics will never get stuck. The parameters are the start memory address and the start sector address: (i) The start memory address is page aligned, (ii) the page to be copied does neither overlap with device domains, (iii) nor with the memory of the C0 kernel machine, and (iv) it lies within the virtual address space, (v) the start sector address is page aligned, i.e. dividable by 8, and (vi) lies outside the boot region of the hard disk.

Formally, we get the following predicate denoting the precondition of both driver functions — reading and writing pages:

$$\begin{aligned}&driver\text{-}precond(ma, sa) \equiv\\ &\qquad \mathsf{PAGE\text{-}SIZE}\ \mathsf{dvd}\ ma\\ &\wedge\quad (ma + 2^{12} - 4) < \mathsf{device\text{-}border}\\ &\wedge\quad \mathsf{ZFP} \cdot \mathsf{PAGE\text{-}SIZE} \leq ma\\ &\wedge\quad px(ma) < \mathsf{ZFP} + \mathsf{VUSPACE}_{\text{pg}}\\ &\wedge\quad 8\ \mathsf{dvd}\ sa\\ &\wedge\quad \mathsf{BOOT}_{\text{pg}} \leq sa \div 8\\ &\wedge\quad sa \div 8 < \mathsf{BOOT}_{\text{pg}} + \mathsf{SWAP\text{-}SIZE}_{\text{pg}}\end{aligned}$$

The semantics of the XCalls for the driver is simple: the write function copies data from the extended memory to the extended swap, and the read function has the reversed effect.

According to Section 3.5 an XCall declaration consists of a triple, defining the types for the input and output parameters, and the semantics of the XCall. For the write case we get:

- **Input parameter types.** The input parameters are equal to the corresponding C0 call, i.e. we have two parameters of type Int.
- **Output parameter types.** The XCall does not alter any global variables, and returns with a value of type Boolean.
- The XCall semantics function takes as arguments the evaluated list of parameters of the XCall and the current extended state (remember that expressions evaluate to values of type contents, which are small memories of basic values). If the parameters fulfill the driver preconditions the C0 constant True and an updated extended state is returned. Otherwise, the computation gets stuck.

```
$xsem_{\text{wdriver}}(args, xpfh) \equiv$
  **let**
    $sa = (args!0)(0)$
    $ma = (args!0)(1)$
    $pfhX' = pfhX[\mathsf{swap} := pfhX.\mathsf{swap}($
      $adjust_{\text{swap}}(sa) := pfhX.\mathsf{mem}(adjust_{\text{mem}}(ma)))]$
  **in**
    **if** $driver\text{-}precond(ma, sa)$ **then** $\lfloor([\mathsf{True}], pfhX')\rfloor$
    **else** $\bot$
```

  We denote the XCall names of the read and write functions with $\mathsf{writePage}_{\text{x}}$ and $\mathsf{readPage}_{\text{x}}$.

Formally, the declaration for the write XCall is given by:

$$\mathsf{writePage}_{\text{xdecl}} \equiv ([\mathsf{Int}, \mathsf{Int}], [\mathsf{Boolean}], \mathsf{xsem}_{\text{wdriver}})$$

For the read case, the input and output parameters are identical to the write case. The semantics $xsem_{\text{rdriver}}$ only differ by the definition of the successor state $pfhX'$. Now, the data is copied from the extended swap to the extended memory component: $pfhX[\mathsf{mem} := pfhX.\mathsf{mem}(adjust_{\text{mem}}(ma) := pfhX.\mathsf{swap}(adjust_{\text{swap}}(sa)))]$. Formally, the declaration for the read XCall is given by:

$$\mathsf{readPage}_{\text{xdecl}} \equiv ([\mathsf{Int}, \mathsf{Int}], [\mathsf{Boolean}], xsem_{\text{rdriver}})$$

4.3.4 Embedding the page-fault handler

In this section we first state the correctness theorem of the C0 driver and then prove its correctness by instantiating the XCall theory developed in Section 3.5. In short, we have to prove implementation correctness against the XCall specification. Finally, we show how the theorem is instantiated to be applied for the kernel verification.

Instantiating the theory We formulate and prove the correctness of the driver implementation against the XCall specification by applying the theory developed in Section 3.5. We first have to define the mapping of function names and the specification map:

$$\begin{array}{ll} \mathit{fns}_{\mathrm{driver}} \equiv & \mathit{specMap}_{\mathrm{driver}} \equiv \\ \quad \{\mathsf{writePage}_{\mathrm{x}} \mapsto \lfloor \mathsf{writePage}_{\mathrm{i}} \rfloor, & \quad \{\mathsf{writePage}_{\mathrm{x}} \mapsto \lfloor \mathsf{writePage}_{\mathrm{idecl}}, \mathsf{writePage}_{\mathrm{xdecl}} \rfloor, \\ \quad\ \mathsf{readPage}_{\mathrm{x}} \mapsto \lfloor \mathsf{readPage}_{\mathrm{i}} \rfloor\} & \quad\ \mathsf{readPage}_{\mathrm{x}} \mapsto \lfloor \mathsf{readPage}_{\mathrm{idecl}}, \mathsf{readPage}_{\mathrm{xdecl}} \rfloor\} \end{array}$$

Now, we instantiate the theory as follows (left column: parameter name, middle column instantiation):

α	pfhX_T	The type of the extended state.
$\mathit{xconsis}_{\mathrm{ex}}$	$\mathit{xconsis}_{\mathrm{driver}}$	The abstraction of the extended state. According to the XCall theory we have to discharge its validity $\mathit{valid}_{\mathrm{ex}}(\mathit{xconsis}_{\mathrm{ex}}, \mathsf{ZFP})$ (cf. Section 3.5). Thus, we have to show that the abstracted memory does not intersect with the memory of the C0 kernel machine and that the hard disk is stable whenever consistency holds. The latter condition follows simply from the predicate *initial-disk* which is part of swap consistency. The first condition holds under the assumption that the heap memory is always bounded by ZFP, since ZFP is the first page of the abstraction. This assumption is therefore included to the correctness theorem described below.
id_X	0	The identifier of the controlled device; in our case the hard disk.
fns	$\mathit{fns}_{\mathrm{driver}}$	Mapping of XCall names to names of functions implementing them.
$\mathit{specMap}$	$\mathit{specMap}_{\mathrm{driver}}$	Mapping containing implementation semantics.

Stating the Simulation The theorem follows the form of the XCall compiler correctness theorem (Theorem 9). The preconditions can be summarized as follows: (i) The intermediate C0 machine and the assembly machine are in a valid state (enabling application of compiler correctness, cf. Section 3.3.7), (ii) the heap consumption of the final state of the extended computation is bounded by *max-address* (i.e. also the heap consumption in each step, cf. Section 3.3.8), (iii) *max-address* itself is bounded by the address of the zero-filled page, (iv) the static estimated stack-size consumption is within the available memory space (cf. Section 3.3.8), (v) the code of the extended machine does neither contain inline assembly code, nor (vi) any XCall invocation except of $\mathsf{writePage}_x$ and $\mathsf{readPage}_x$, (vii) XCall consistency holds, (viii) the program rest of the extended machine consists of a single function call (that is the granularity at which properties are transfered), and (ix) a valid extended computation of k steps exists.

Once the preconditions are granted, we show that for all valid sequences a VAMP assembly with devices computation exists, such that the following postconditions hold: (i) It exists a new intermediate C0 machine, such that consistency again holds, (ii) the VAMP assembly with devices computations fulfills the requirements of the ISA-assembly translation (cf. Section 3.1.5), (iii) the assembly and C0 configurations are valid again, and (iv) the VAMP assembly with devices computations is pure, i.e. only the hard disk is accessed (cf. Section 3.2).

Moreover, we have an additional postcondition, which concerns are all other devices than the hard disk. Because, the code only accesses the hard disk, the computation of other devices is not influenced by the simulated computation of the processor. Still, due to input from the external environment, the configuration of the devices may have changed. Thus, the final state of any device can be computed by ignoring all steps of the processor and of the hard disk. Note, that this final postcondition follows directly from purity by applying Theorem 3.

Formally, we get the following theorem:

Theorem 11 (Driver XCalls Compiler Correctness)

$$
\begin{array}{ll}
 & \mathit{max\text{-}address} \le \mathsf{ZFP} \\
\wedge & \mathit{valid\text{-}C0}(\mathit{tenv}, \mathit{pt}_i, c) \\
\wedge & \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asmd}.\mathsf{proc}, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm})) \\
\wedge & \mathsf{heap\text{-}base} + \mathit{heap\text{-}size}(c_X{}'.c, \mathit{pt}_i) < \mathit{max\text{-}address} \\
\wedge & \mathit{max\text{-}address} < \mathsf{device\text{-}border} \\
\wedge & \mathit{sz} \in \mathit{ub\text{-}costs}(\mathit{tenv}, \mathit{pt}_i, \mathit{fn}_i) \\
\wedge & \mathit{sz} + \mathit{stack\text{-}start}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm}) + \mathit{stack\text{-}size}(\mathit{tenv}, c) \le \mathsf{heap\text{-}base} \\
\wedge & \mathit{valid\text{-}prop}_{\mathit{code}}(c_X.\mathsf{c}.\mathsf{prog}, \mathit{pt}_x, (\lambda s.s \ne \mathit{Asm}(\dots))) \\
\wedge & \mathit{valid\text{-}prop}_{\mathit{code}}(c_X.\mathsf{c}.\mathsf{prog}, \mathit{pt}_x, \\
 & \quad (\lambda s.s = \mathit{XCall}(\mathit{fn}_x, \dots) \implies \mathit{fn}_x \in \mathit{dom}(\mathit{specMap}_{\mathit{driver}}))) \\
\wedge & \mathit{xconsis}(\mathit{tenv}, \mathit{pt}_i, c, \mathit{alloc}, \mathit{asmd}, \mathit{pt}_x, c_X, \mathit{xpt}, \mathit{fns}_{\mathit{driver}}, \mathit{specMap}_{\mathit{driver}}, \mathit{rest}, \mathit{id}_X) \\
\wedge & c_X.\mathsf{c}.\mathsf{prog} = \mathsf{SCall}(\dots, \mathit{fn}_i, \dots) \\
\wedge & \delta_{\mathit{cx}}{}^k(c_X, \mathit{tenv}, \mathit{pt}_i, \mathit{xpt}) = \lfloor c_X{}' \rfloor \\
\implies & (\forall \mathit{seq} \in \mathit{Seq}_V\,.\, \exists c', \mathit{alloc}', T. \\
 & \quad \mathit{xconsis}(\mathit{tenv}, \mathit{pt}_i, c', \mathit{alloc}', \mathit{asmd}^{\mathit{seq},T}, \\
 & \quad\quad \mathit{pt}_x, c_X{}', \mathit{xpt}, \mathit{fns}_{\mathit{driver}}, \mathit{specMap}_{\mathit{driver}}, \mathit{rest}, \mathit{id}_X) \wedge \\
 & \quad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{dyn}}(\mathit{asmd}, \mathit{code\text{-}base}, \\
 & \quad\quad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm}), \mathit{seq}, T) \wedge \\
 & \quad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asmd}^{\mathit{seq},T}.\mathsf{proc}, \mathit{code\text{-}base}, \\
 & \quad\quad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm})) \wedge \\
 & \quad \mathit{valid\text{-}C0}(\mathit{tenv}, \mathit{pt}_i, c') \wedge \\
 & \quad \mathit{pure}(\mathit{asmd}, \mathit{seq}, T, \mathit{id}_X) \wedge \\
 & \quad\ (\forall \mathit{id}_Y \ne 0\,.\quad \mathit{asmd}^{\mathit{seq},T}.\mathsf{devs}(\mathit{id}_Y) = \\
 & \quad\quad\quad\quad \mathit{asmd}^{\pi(\mathit{seq},\mathit{id}_X),\mathit{step\text{-}nr}(\mathit{seq},0,T)}.\mathsf{devs}(\mathit{id}_Y)))
\end{array}
$$

Proving implementation correctness The correctness of the theorem follows almost immediately by instantiating Theorem 9 as described above and discharging its first assumption. This assumption states that the XCalls writePage and readPage are implemented correctly.

Formally, we have to prove that the predicates $\mathit{correct}_{\mathrm{xcall}}(\mathit{specMap}, \mathsf{writePage}_{\mathrm{x}}, 0)$ and $\mathit{correct}_{\mathrm{xcall}}(\mathit{specMap}, \mathsf{readPage}_{\mathrm{x}}, 0)$ hold (cf Section 3.5). For the write case this unpacks to the following Lemma:

Lemma 35 (Correctness of writePage implementation)

$$
\begin{array}{ll}
 & \mathit{valid\text{-}C0}(\mathit{tenv}, \mathit{pt}_i, c) \\
\wedge & \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asmd}.\mathsf{proc}, \mathit{code\text{-}base}, \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm})) \\
\wedge & \mathsf{heap\text{-}base} + \mathit{heap\text{-}size}(c_X{}'.c, \mathit{pt}_i) < \mathit{max\text{-}address} \\
\wedge & \mathit{max\text{-}address} < \mathsf{device\text{-}border} \\
\wedge & \mathit{sz} \in \mathit{ub\text{-}costs}(\mathit{tenv}, \mathit{pt}_i, \mathsf{the}(\mathit{fns}_{\mathit{driver}}(\mathit{fn}_x))) \\
\wedge & \mathit{sz} + \mathit{stack\text{-}start}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm}) + \mathit{stack\text{-}size}(\mathit{tenv}, c) \leq \mathsf{heap\text{-}base} \\
\wedge & \mathit{xconsis}(\mathit{tenv}, \mathit{pt}_i, c, \mathit{alloc}, \mathit{asmd}, \mathit{pt}_x, c_X, \mathit{xpt}, \mathit{fns}_{\mathit{driver}}, \mathit{specMap}_{\mathit{driver}}, \mathit{rest}, \mathit{id}_X) \\
\wedge & c_X.\mathsf{c}.\mathsf{prog} = \mathsf{XCall}(\mathsf{writePage}_x, \mathit{lvars}, \mathit{exps}_p) \\
\wedge & \delta_{cx}(c_X, \mathit{tenv}, \mathit{pt}_i, \mathit{xpt}) = \lfloor c_X{}' \rfloor \\
\Longrightarrow & (\forall \mathit{seq} \in \mathit{Seq}_V(\mathit{id}_X)\,.\,\exists c', \mathit{alloc}', T. \\
 & \quad \mathit{xconsis}(\mathit{tenv}, \mathit{pt}_i, c', \mathit{alloc}', \mathit{asmd}^{\mathit{seq},T}, \\
 & \quad\quad \mathit{pt}_x, c_X{}', \mathit{xpt}, \mathit{fns}_{\mathit{driver}}, \mathit{specMap}_{\mathit{driver}}, \mathit{rest}, \mathit{id}_X) \wedge \\
 & \quad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{dyn}}(\mathit{asmd}, \mathit{code\text{-}base}, \\
 & \quad\quad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm}), \mathit{seq}, T) \wedge \\
 & \quad \mathit{isa\text{-}asm\text{-}precond}_{\mathit{init}}(\mathit{asmd}^{\mathit{seq},T}.\mathsf{proc}, \mathit{code\text{-}base}, \\
 & \quad\quad \mathit{code\text{-}size}(\mathit{tenv}, \mathit{pt}_i, c.\mathsf{mf}.\mathsf{gm})) \wedge \\
 & \quad \mathit{valid\text{-}C0}(\mathit{tenv}, \mathit{pt}_i, c') \wedge \\
 & \quad \mathit{pure}(\mathit{asmd}, \mathit{seq}, T, \mathit{id}_X))
\end{array}
$$

The proof can be conducted at the level of the C0 with inline assembly semantics $\rightarrow_{\mathrm{c+ad}}$ introduced in Section 3.4 (or alternatively by manually applying the reordering theory, as conducted formally). Using Theorem 8 results are propagated down to VAMP assembly with devices. Thus, we can easily switch between reasoning about the execution of C0 code and inline assembly without interference.

The first statement to execute is the C0 invocation of the C0 function $\mathsf{writePage}_\mathsf{i}$. It establishes a new function frame and copies the parameters to it. The execution of a function call only gets stuck if either the function is not declared in the given procedure table or if the parameter evaluation gets stuck. Both can not occur here: $\mathit{specMap}_{\mathrm{driver}}$ ensures that the procedure table implements the function, and since the XCall invocation leads to a valid state, also the parameter evaluation must be valid. We denote the new C0 state with c_1.

The next statement is the inline assembly code of the driver. Therefore, we switch to assembly semantics (by applying the corresponding rule in the semantics) and instantiate Theorem 10, stating correctness of the assembly driver. The assembly state obtained after switching is denoted by asm_1 and is guaranteed to be consistent to the C0 configuration after the call. The preconditions of the theorem are discharged as follows:

- The precondition $\mathit{driver\text{-}precond}_{\mathrm{hd}}$ on the hard disk follows directly from the definition of the extended consistency predicate defined above. Moreover, according to the first rule of the transition system, we can assume that the hard disk did not change during the execution of the SCall statement.

- For the assembly machine we have the following assumptions:

$\mathit{valid\text{-}asm}(asm_1)$	Validity of the assembly machine is guaranteed by compiler correctness.
$asm_1.\mathsf{spr!MODE} = 0$	System mode is ensured at the beginning by the precondition $\mathit{isa\text{-}asm\text{-}precond}_{\text{init}}$ and preserved by the compiler, since compiler correctness ensures that no special purpose registers are altered.
$asm_1.\mathsf{spr!SR} = 0$	Interrupts stay disabled for the same reason as the status register.
$asm_1.\mathsf{dpc} + \mathsf{d\text{-}length} < \mathsf{device\text{-}border}$	Compiler correctness ensures that the program counter is still within the overall C0 code. From the memory layout we easily deduce that the code is smaller than the heap-base. For the latter we know $\mathit{max\text{-}address} < \mathsf{device\text{-}border}$.
$\mathit{to\text{-}instr\text{-}list}(asm_1.\mathsf{mm}, asm_1.\mathsf{dpc}, \mathsf{d\text{-}length}) = \mathit{d\text{-}code}(id_{D\text{hd}}, sa_{\text{off}}, ma_{\text{off}})$	We have to show, that the code pointed to in the new configuration is the one stated in the corresponding statement. This follows from an application of Lemma 22.

Finally, we have to discharge the assumptions on the call parameters:

$\mathit{valid\text{-}ad}(\mathit{i2n}(asm_1.\mathsf{gpr!topIm}) + ma_{\text{off}})$	Validity of the parameter addresses is ensured by the predicate $\mathit{valid\text{-}c0}$.
$\mathit{valid\text{-}ad}(\mathit{i2n}(asm_1.\mathsf{gpr!topIm}) + sa_{\text{off}})$	Same as above.

$ma = i2n(asm_1.\mathsf{mm}(i2n(asm_1.\mathsf{gpr}!\mathsf{toplm})+ma_{\mathrm{off}}))$	The semantics of the SCall ensures, that parameters passed in the function call are correctly evaluated and stored in the function frame. Compiler correctness ensures that the function frame is correctly stored at the top-most stack frame, whose address is in turn stored in register toplm.
$sa = i2n(asm_1.\mathsf{mm}(i2n(asm_1.\mathsf{gpr}!\mathsf{toplm}) + sa_{\mathrm{off}}))$	Same as above.
$\mathit{valid\text{-}ad}(ma)$	Validity of the memory addresses follow from the precondition $\mathit{driver}_{\mathrm{pre}}$ of the XCall on the parameters. These parameters are the same evaluated expressions passed to the SCall and stored in the stack of the consistent assembly machine.
$sa + 8 < S$	Same as above.

After the application of Theorem 10 we obtain a new VAMP assembly with devices state $asmd_2$, in which the post conditions of the theorem hold. Next, we want to switch again back to the C0 semantics to obtain a new and consistent intermediate C0 configuration. For that, we apply the third rule of the transition relation $\rightarrow_{\mathrm{c+ad}}$. The first condition requires the program rest of the original C0 machine to point to an assembly statement. This follows from the implementation of the driver. The last condition, requires that the assembly computation is pure, which is a postcondition of Theorem 10. We still have to discharge the second condition, i.e. to prove that the function $\mathit{co\text{-}asm\text{-}update}(tenv, c_1, asm_1, asm_2, [])$ returns with some updated C0 state c_2. Note, that the set of altered variables is chosen to be empty, since the inline assembly driver does not change any global variables during execution. The preconditions of the function require that neither the code, nor memory or any registers used by the compiler were manipulated during the execution of assembly. All three conditions are guaranteed by the postcondition $\mathit{driver\text{-}post}_{\mathrm{proc}}$ of Theorem 10.

At this stage, consistency on the extended component can already be established. The proof of this claim is straightforward, since the XCall semantics mimics exactly the copy of a sector to the hard disk.

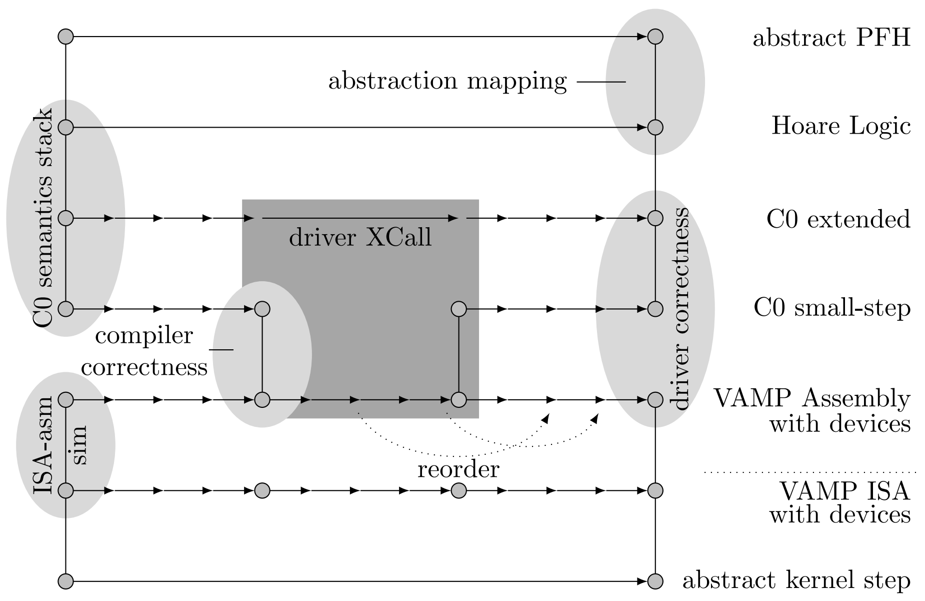


Figure 4.5: Putting It All Together – Correctness of the Page-Fault Handler

Finally, the last C0 statement — the return from the function call — is executed. This establishes again XCall consistency on the code (the proof is tricky, since it involves reasoning about prefix equivalence of statements). By applying the first rule of the transition system (or, equivalently, by applying the extended compiler correctness theorem) we deduce that the disk state did not change. Moreover by compiler correctness, we can also infer that the memory region, which is abstracted to the extended component, also stayed unchanged. Thus, also the extended consistency still holds. Still, we have to show that the memory of the intermediate C0 machine c_2 and of the extended machine $c_X{}'$ are equal. This holds, since the only write access of the driver to the memory is when the return value is written back. But this operation is also mimicked by the corresponding XCall semantics. This establishes consistency again.

Instantiating the theorem for the page-fault handler Page-fault handler correctness is ultimately expressed as a simulation between VAMP ISA with devices and an abstract transition system, where neither page-faults nor swap memory are visible (cf. Figure 4.5, below the dashed line). In the former model, one of the devices is instantiated to the hard disk. Page-fault handler verification, however, is conducted on many different semantical layers. On the one hand, switches between user and system mode forces us to toggle also between VAMP ISA and VAMP assembly. On the other hand, there are code switches between assembly and C0. The C0 language stack with its extension to XCalls and devices is used to separate verification goals and apply on each level the adequate technique. Finally, everything is pushed down to VAMP ISA with devices.

The overall proof structure is depicted in Figure 4.5. First the page-fault handler code — which is enriched with XCalls — is verified against an abstract page-fault handler specification[2] in the extended C0 Hoare logic. Subsequently, this result is lifted via the C0 language stack down to the extended C0 small-step semantics. Using Theorem 11 page-fault handler correctness is transferred to VAMP assembly with devices and finally expressed at the level of VAMP ISA with devices by a simple straightforward application of Theorem 1 (all preconditions are guaranteed by the simulation proven above).

For the transfer from extended C0 semantics down to VAMP assembly with devices, Theorem 11 is instantiated as follows:

- The intermediate C0 machine is instantiated to the kernel C0 machine containing the page-fault handler, i.e. the page table pt_{i}, the type environment *tenv* and the *max-address* are set accordingly.
- The SCall is set to the page-fault handler.

[2]Concrete doubly-linked lists are for example abstracted to Isabelle lists, etc.

Part II

Proving the Correctness of Client / Server Software

Chapter 5

Specifying an Operating System

In this chapter we introduce a formal specification of the operating system SOS developed in Verisoft. The obtained model SOS* [Bog08] is a system of distributed user applications which can communicate with each other and the operating system. The latter, provides user applications with a set of so called system calls, i.e. services which for example ensure access to inter process communication or to the portmapper. In a nutshell SOS* serves:

- *Downwards* as specification for the underlying operating system implementation, and
- *upwards* as programming model for user applications running on top of the SOS.

For this thesis, we are only interested in the second goal, since we want to apply SOS* to verify client/server applications running in SOS.

We start in the next section with an overview on the SOS. We proceed in Section 5.2 with a formal definition of a subset of SOS*. Finally, in Section 5.3, to ease reasoning about the distributed system, we introduce a theory of non-interfering system calls. This theory enables us to verify parts of the distributed computation sequentially.

5.1 Background

As in many other recent projects, we have split our operating system into a part running in system mode, i. e. the VAMOS micro kernel, and a part running in user mode, i. e. the SOS. In this subsection, we summarize the properties of the VAMOS micro kernel and briefly describe the SOS implementation.

5.1.1 VAMOS

The VAMOS micro kernel provides isolation of processes by means of virtual memory. Processes can be dynamically created and killed. They may communicate via synchronous inter-process communication (IPC). This communication can be controlled via permissions assigned to each pair of communication partners. IPC messages may have arbitrary size. VAMOS is equipped with a priority-based scheduler. Processes may register as user level device drivers. Processes that are registered as user level device drivers (for a certain device), receive interrupt notifications by means of IPC messages. Finally, the kernel maintains different privilege levels and thereby facilitates the implementation of user mode operating systems.

5.1.2 SOS

The operating system SOS is implemented on top of the VAMOS micro kernel. It is a process running in user mode. The kernel initially starts the SOS as the only process. It assigns the highest scheduling priority to the SOS and marks it as privileged. Therefore the SOS is enabled to register device drivers, to start and stop user processes, or to assign memory. In fact, most of the kernel calls provided by VAMOS require the calling process to be a privileged process. Unprivileged processes, which we will call *user applications*, are restricted to IPC and a small number of IPC related kernel calls. However, by means of *SOS calls*, user applications are provided with more versatile and more powerful calls. For example, the kernel call that allows to create a new process is substituted by an SOS call that allows to start a new application from an executable file. These SOS calls are transmitted to and answered by the SOS via kernel IPC calls.

The SOS itself is implemented in C0 (with special calls to kernel services which also can be abstracted via the concept of XCalls). Similarly, also the user processes are instantiated to C0 machines.

In order to support user applications calling the SOS, a C0 library is provided that wraps the necessary IPC calls and hides the difference between kernel calls and SOS calls. Figure 5.1 shows the user applications' limited access to the kernel. Furthermore, it shows how the SOS serves as intermediate layer between user applications and kernel. Note that, for performance reasons, a number of kernel calls remain accessible to user applications. For example, routing IPC-messages through the SOS would dramatically slow down IPC, double the number of context switches, and increase the SOS's latency and memory consumption.

The SOS provides 31 calls, allowing user applications to: 1. manage different users, 2. interact with a file system, 3. communicate with the outside world, 4. handle applications, 5. locate and register remote procedure services, and 6. interact with virtual terminals.

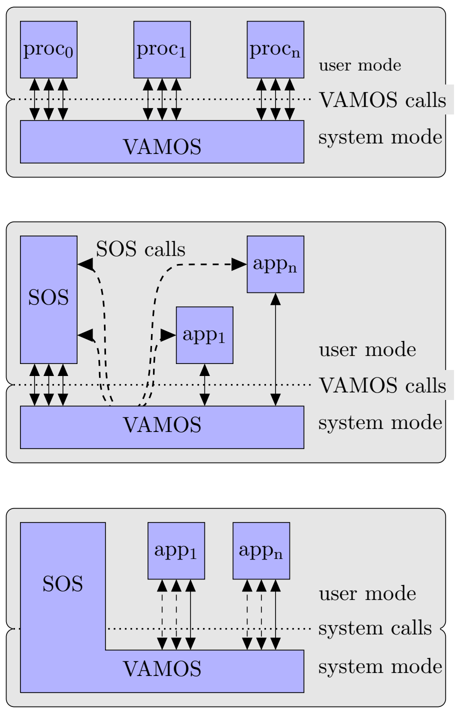


Figure 5.1: SOS Calls. 1. User processes solely rely on VAMOS calls. 2. In the presence of the SOS, user applications are restricted to a few VAMOS calls but (via IPC) they may use SOS calls. 3. User applications cannot see the difference between VAMOS calls and SOS calls. Thus, from the application point of view, the SOS process and the VAMOS micro kernel melt together. The resulting (single) operating system provides services in terms of system calls.

5.2 Specification of the SOS

In this section we describe SOS*, a model of a whole computer system. We start out with an overview of the main SOS* components and then present an exact definition of each of these components.

Note that the specification we present here is only an extract of Bogan's comprehensive work [Bog08] and was first published in [ABP09]. Many aspects (of the complete specification) are either left out or simplified. For example, within this document, we are not formalizing any of the SOS calls related to network communication, file access, or terminal I / O. Furthermore, for IPC calls, we do not consider the combined send-receive operation and neglect the IPC rights.

5.2.1 Notation

The goal of this section is to describe a formal model on top of which RPC is implemented, specified and verified. Since RPC correctness is not formalized in Isabelle/HOL, we should stay as close as possible to the original SOS* formalization in [Bog08]. Therefore, the notation used in the remainder of this chapter differs slightly from the one used previously in the thesis. In the following we summarize these differences.

Abstract Data types. Elements of abstract data types are written without brackets. For example, we write for the list constructor $\mathsf{CON}\ x\ xs$ instead of $\mathsf{CON}(x, xs)$.

Sequences. Sequences are not modeled any more as functions. Rather, for any type T and natural number n, we denote by T^n the sequence consisting of n elements of type T. Furthermore, we denote the type of arbitrary (finite and infinite) sequences of elements of type T by T^*. We assume that elements of a sequence can be enumerated (starting from 0). Access to the i-th element of a sequence x is denoted by x^i.

Functions. The undefined return value *undef* is made explicitly visible in sets and denoted by ϵ. So, we use, for example, $g \in \mathbb{Z} \to \mathbb{N} \cup \{\varepsilon\}$ to declare a function that would otherwise only be defined for a subset of $\mathbb{Z}$. As we use this 'type extension'frequently, we write T_ε as a shorthand for $T \cup \{\varepsilon\}$.

Sets. We introduce restricted sets of naturals and integers denoted by $\mathbb{N}_{32} = \{0, \dots, 2^{32} - 1\}$, $\mathbb{N}^+_{32} = \mathbb{N}_{32} \setminus \{0\}$, and $\mathbb{Z}_{32} = \{-2^{31}, \dots, 2^{31} - 1\}$. In rare cases we use the Hilbert Choice Operator to select an arbitrary element from a given non-empty set. We denote the Hilbert Choice Operator by **s-el**.

5.2.2 Overview on the Formal Model

Formally, SOS* is defined as a transition system:

$$\text{SOS}^\star = (S, \Delta, R, \dots).$$

Where each component has the following meaning:

S	The set of possible configurations (the SOS* state space).
$\Delta \subset S \times S$	The transition relation.
R	Characterizes the set of valid SOS* runs.

SOS* is intended to be used as a programming model for communicating user applications. In SOS*, we choose not to restrict the types of user applications that may be verified to a particular programming language. Instead, we incorporate user applications in the form of I / O automata [Lyn96]:

$$\text{APP}^\star = (S_p, \Sigma_p, \Omega_p, \delta_p, \omega_p, \dots).$$

Where each component has the following meaning:

S_p	The set of possible configurations (the application state space).
Σ_p	The input alphabet (system call results).
Ω_p	The output alphabet (system calls).
$\delta_p \in (S_p \times \Sigma_p \cup \{\varepsilon\}) \rightarrow S_p$	The transition function.
$\omega_p \in (S_p) \rightarrow (\Omega_p \cup \{\varepsilon\})$	Computes the application output for a given state.

Describing user applications as automata has the advantage that the abstraction can be easily instantiated by different machine types (e. g. assembly- or C0 programs). A different machine type does not change the global transition system, as long as the new machine type complies with the (interface) alphabets Σ_p and Ω_p. For SOS* that means that the alphabets Σ_p and Ω_p must be well defined. The remaining types and functions of APP*, however, may be SOS* parameters. Hence, we get the following (updated) definition of SOS*:

$$\text{SOS}^\star(S_p, \delta_p, \omega_p, \dots) = (S, \Delta, R, \Sigma_p, \Omega_p, \dots).$$

Note, although different machine types are supported, S_p, δ_p, and ω_p have to be fixed throughout a model run. That means all user applications share the same S_p, δ_p, and ω_p (which still might cover assembly- *and* C0 programs). Hence, the subscript "p" (as in ω_p) is not an index but belongs to the function name.

After introducing the overall concept of SOS*, we will now elaborate on each of its components.

5.2.3 State Space

User applications perceive their environment, i. e. the underlying hardware and system software as well as other user applications, by means of system calls. In order to correctly formalize the behavior of these calls, the SOS* state space contains data structures that represent the user visible part of the environment. In this subsection we describe selected components of S. The complete SOS* model has many more components. Here, we describe only those parts that are relevant for RPC.

Users

SOS is a multi-user operating system. It allows different registered users to log in. A user is thereby referred to by his user id. In SOS, all registered users are stored in the user data base.

In SOS*, we represent user ids by numbers. User ids have the type $uid\_t \subset \mathbb{N}_{32}$. The state-space component udb contains all registered users:

$$\mathsf{udb} : pow(uid\_t).$$

User Applications

The SOS supports communicating user applications. Previously, we already explained how user applications can be modeled as I / O automata. Now we describe how their representation is integrated into the SOS* state space.

User applications are manifested in the SOS* state space in three ways. They are represented by:

- a local state, i. e. the application's internal configuration (e. g. the assembly- or C0 configuration),
- a number of process-related data structures, i. e. bookkeeping data structures about the user process maintained by the kernel, and
- a number of application-related data structures, i. e. bookkeeping data structures maintained by the SOS.

Local State. From the kernel's point of view, the SOS and the user applications are user processes. The maximum number of simultaneously running user processes is denoted by $\text{MAXPROC} \in \mathbb{N}^+_{32}$. The kernel uses process identifiers (PIDs) to refer to specific processes. In SOS*, the set of all PIDs is represented by $pid\_t = \{1, \ldots, \text{MAXPROC}\}$. Here, the constant $\text{OSPID} \in pid\_t$ denotes the PID of the SOS process. The local states of all user applications are stored in the process data base pdb. This state-space component is a function that maps PIDs to process states. Currently unassigned PIDs are mapped to ε. Thus:

$$\mathsf{pdb} : pdb\_t.$$

Where:

$$pdb\_t = pid\_t \setminus \{\text{OSPID}\} \to S_p \cup \{\varepsilon\}.$$

Note that although the SOS is a user process, it is not a user application and, therefore, invisible in SOS*. Hence, OSPID is excluded from the domain of pdb.

Kernel Data Structures about User Processes. In SOS*, there are a number of data structures that are, in the implementation, maintained by the kernel. In the complete model, these kernel data structures are combined in the (SOS*) state space component kds. Here, we discuss only one of these kernel data structures, i. e. the handle data base.

User processes exclusively identify each other using so-called handles. Handles are local names for PIDs. On a per-process basis, the kernel maintains the mapping between *handles* and *PIDs* . This mapping is called the handle data base. As we will explain later, unless a process has a handle for another process, there is no way for the earlier to approach the latter. Thus, the indirection through handles provides for better control of information flow. In SOS*, the handle data base is represented by the kds component hdb:

$$\mathsf{hdb} : \mathit{pid\_t} \times \mathit{hn\_t} \to \mathit{pid\_t}_{\perp}.$$

Here, $\mathit{hn\_t}$, with $\mathit{hn\_t} \subset \mathbb{N}_{32}$, represents the set of possible handles. Currently unassigned handles are mapped to ε. There are a number of special handles. For the work at hand, only the pseudo handle identifying no process is relevant. In SOS* this handle is denoted by HN-NONE $\in \mathit{hn\_t}$.

As mentioned earlier, the complete SOS* model contains more kernel data structures. There, handles are, for example, accompanied by communication rights. These rights provide additional means to fine tune the inter-process communication. For the work at hand, however, the handle data base is the only relevant kernel data structure. Thus, here, the state space component kds has only one record field:

$$\mathsf{kds} : \mathit{kds\_t}.$$

Where:

$$\mathit{kds\_t} = (\mathsf{hdb} : \mathit{pid\_t} \times \mathit{hn\_t} \to \mathit{pid\_t}_{\perp}, \dots).$$

SOS Data Structures about User Applications. The SOS keeps track of all user processes and adds rights management and access control based on users. It thereby establishes the concept of user applications. For each user application, the SOS maintains a certain amount of information. It stores, for example, which user started a particular application and whether a certain application has access to the screen. In SOS*, the information about a single user application is represented by an instance of type *app_t*. Among other things, this type contains the field owner representing the user owning the application:

$$\mathit{app\_t} = (\mathsf{owner} : \mathit{uid\_t}, \dots).$$

Again, in the complete SOS* model, *app_t* contains more fields. The information about all user applications is assembled in the application data base. In SOS*, this data base is represented by the function adb, which maps handles to the associated information. Currently unassigned application handles are mapped to ε:

$$\mathsf{adb} : \mathit{hn\_t} \to \mathit{app\_t}_{\perp}.$$

Portmapper

The SOS provides infrastructure for so-called RPCs. RPCs allow one application, the client, to take advantage of some service provided by another application, the server. Here, a service is specified by an interface name and a procedure name. At compile time, clients know the names of the services they intend to call. However, the location of this service, i. e. the handle of the providing application, is unknown at that time. Hence, we need a run-time mapping of service names to service providers. This mapping is called *portmapper data base.*[1]

In SOS*, a service name is represented by the type *service_t*. Here, a service name is the combination of the interface id $iid\_t \subset \mathbb{N}_{32}$ and a procedure id $prcid\_t \subset \mathbb{N}_{32}$, i. e. $service\_t = iid\_t \times prcid\_t$.

Based on service names, the portmapper data base comprises:

- serv a mapping between interface ids and the handles of the providing servers,
- reg a set of registered services, and
- known, the set of known services.

In SOS*, the portmapper data base is represented by the state-space component *pmdb* of the following record type:

$$\mathsf{pmdb} : pmdb\_t.$$

Where:

$$pmdb\_t = \left(\mathsf{serv} : iid\_t \to hn\_t_\perp, \mathsf{reg} : pow(service\_t), \mathsf{known} : pow(service\_t) \right).$$

Note that we need the component known because a portmapper usually only supports a set of well-known services. In addition, note that (supported) interfaces that are not served, are mapped to ε. Finally, servers can only register for one interface. Such an interface, however, may contain several procedures serving different purposes.

Summing Up

Now, collecting all pieces of the (reduced) SOS* state space, S is defined as follows:

$$\begin{aligned} S = \ (\quad & \mathsf{udb} : pow(uid\_t), \\ & \mathsf{pdb} : pdb\_t, \mathsf{kds} : kds\_t, \mathsf{adb} : hn\_t \to app\_t_\perp, \\ & \mathsf{pmdb} : pmdb\_t, \\ & \ldots \\ ). & \end{aligned}$$

[1] Currently, our portmapper implementation only supports local inquiries and instead of mapping services to IP addresses and port numbers, it maps services to handles. This could be easily changed but for now this simplified version suffices to serve our needs.

5.2.4 Alphabets

In SOS*, we choose to represent system calls and system-call results in a machine independent form, i. e. the alphabet Ω_p and Σ_p. In this section, we describe all those elements of Ω_p and Σ_p that are relevant in the context of this paper.

The alphabet Ω_p contains the abstract representations of the available system calls. In the work at hand, Ω_p only contains:

- representations for the portmapper calls to register, look up, and unregister services (REG, LUP, and UNREG),
- representations for sending and receiving IPC messages (SND and RCV), and
- representations for undefined SOS- and kernel calls (UNDEF-SC and UNDEF-KC).

These calls are represented by the following abstract data type:

$$
\begin{aligned}
\Omega_p = \quad & \textsc{reg} \textbf{ of } \mathit{iid\_t}_\perp \times \mathit{prcid\_t}_\perp \mid \\
& \textsc{lup} \textbf{ of } \mathit{iid\_t}_\perp \times \mathit{prcid\_t}_\perp \mid \\
& \textsc{unreg} \textbf{ of } \mathit{iid\_t}_\perp \times \mathit{prcid\_t}_\perp \mid \\
& \textsc{undef-sc} \mid \\
& \textsc{undef-kc} \mid \\
& \textsc{snd} \textbf{ of } \mathit{hn\_t}_\perp \times \mathit{byte\_t}^* \cup \{\varepsilon\} \times \{\textsc{finite}, \textsc{infinite}\} \mid \\
& \textsc{rcv} \textbf{ of } \mathit{hn\_t}_\perp \times \mathbb{N}_{32} \times \{\textsc{finite}, \textsc{infinite}\} \mid \\
& \ldots
\end{aligned}
$$

Note that we denote all finite timeouts by FINITE and infinite timeouts by INFINITE.[2] An example for a send ipc call would be SND $h\ m\ t$ and for receive RCV $h\ b\ t$. Note, that in this example, we use $m \in \mathit{byte\_t}^* \cup \{\varepsilon\}$ and $b \in \mathbb{N}_{32}$ as abstract representations for messages and buffers, respectively. We choose this representation to be universal enough to support arbitrary process abstraction and at the same time describe the essence of messages and buffers, i. e. a sequence of bytes and a container of a certain size. Finally, note that it may be that a process calls the system but passes along parameter values that do not match the required type (e. g. $id_i \notin \mathit{iid\_t}$). In SOS*, we represent all of these values through the symbol ε. Hence, most of the parameter types are unions of the required type and ε.

The counterpart to Ω_p is Σ_p. The alphabet Σ_p contains the abstract representations of the available system-call results. In the work at hand, Σ_p only contains:

[2]For a number of reasons that are detailed in [Dau08], the scheduler is no longer visible in SOS*. Along with that, there is no precise notion of time. Therefore, we are not able to model timeout situation more precisely than "a certain system call might timeout" or "a certain system call cannot timeout".

- the result messages for a successful portmapper look-up and a successful IPC-receive operation (SUCC-LUP and SUCC-RCV) and
- some general purpose success and error messages (SUCC, ERR, and TIMEOUT).

Again, each of these results is represented by a constructor indicating the type of system-call result and (possibly) an additional parameter:

$$\begin{aligned}\Sigma_p = \quad & \text{ERR} \mid \\ & \text{TIMEOUT} \mid \\ & \text{SUCC} \mid \\ & \text{SUCC-LUP } \textbf{of } \mathit{hn\_t} \mid \\ & \text{SUCC-RCV } \textbf{of } \mathit{hn\_t} \times \mathit{byte\_t}^* \mid \\ & \ldots\end{aligned}$$

Now, no matter which process abstraction was chosen, one can define a function $\omega_p \in S_p \to \Omega_p \cup \{\varepsilon\}$ that maps a process's state to:

- $x \in \Omega_p$, if a particular process wants to call the system, or
- ε, if the processes wants to perform a local computation, i. e. a computation that does not involve calling the system.

If an SOS call was received and treated by the SOS, then the results must be returned to the appropriate application. For returning these results, we simply assume that the local transition function δ_p is defined to take $\Sigma_p \cup \{\varepsilon\}$ as (process-external) inputs, i. e. $\delta_p \in S_p \times \Sigma_p \cup \{\varepsilon\} \to S_p$. Using δ_p, we can define how the (global) SOS* state is changed when a system-call result is returned. As this is a quite common task, we define the function *res* as follows:

$$\mathit{res} \in S \times \mathit{pid\_t} \times \Sigma_p \to S$$

$$\mathit{res}(s, p, r) = s[\mathsf{pdb}(p) := \delta_p(s.\mathsf{pdb}(p), r)].$$

5.2.5 Transition Relation

In this section we define the (reduced) SOS* transition relation. Again, we only define those transitions that are relevant in order to argue about RPC.

Auxiliaries

Resolving handles to PIDs and vice versa will be done quite often. Thus, as a short hand, we define the functions *ph2p* and *pp2h*.

The call $\mathit{ph2p}(s, p, hn)$ inspects the handle data base of p and returns the PID, hn is pointing to. If, for p, the handle hn is not assigned, then ε is returned:

$$\mathit{ph2p} \in S \times \mathit{pid\_t} \times \mathit{hn\_t} \to \mathit{pid\_t}_\perp$$

$$\mathit{ph2p}(s, p, hn) = s.\mathsf{kds}.\mathsf{hdb}(p)(hn).$$

The call $pp2h(s, p_1, p_2)$ returns the handle the process p_1 may use to refer to p_2. If p_2 is unknown to p_1, then ε is returned:

$$pp2h \in S \times pid\_t \times pid\_t \to hn\_t_\perp$$

$$pp2h(s, p_1, p_2) =$$
$$\begin{cases} \textbf{s-el}\{x \mid s.\mathsf{kds.hdb}(p_1)(x)\} & \textbf{if } \exists x \in pid\_t.\ s.\mathsf{kds.hdb}(p_1)(x) = p_2, \\ \varepsilon & \textbf{otherwise}. \end{cases}$$

Note, for $pp2h(s, p_1, p_2)$ to be deterministic, $s.\mathsf{hdb}$ needs to be an injective function. Thus, we require that (for each process) there are no two (different) handles pointing to the same process.

For resolving handles and PIDs from the perspective of the SOS, we additionally provide the following short hands $h2p(s, hn) = ph2p(s, \text{OSPID}, hn)$ and $p2h(s, p) = pp2h(s, \text{OSPID}, p)$.

Transitions

After the preparatory work, we can know specify the individual transitions.

Register a service. The SOS implementation provides a system call that allows user applications to register as server providing a certain service. In SOS*, this call is represented by REG id_i id_p, where id_i and id_p are the interface- and procedure id, respectively.

In order to successfully register a service, a number of preconditions must be met. The predicate $vrega?$ is satisfied, if these preconditions are met. For $vrega?(s, p, id_i, id_p)$ to hold, (id_i, id_p) must be a service known to the portmapper, the calling process p is not yet registered to serve a different interface, and the interface id_i has not been registered by a different process, i. e. $s.\mathsf{pmdb.serv}(id_i) \in \{p2h(s, p), \varepsilon\}$. That is:

$$vrega? \in S \times pid\_t \times iid\_t_\perp \times prcid\_t_\perp \to bool$$

$$\begin{aligned} vrega?(s, p, id_i, id_p) \equiv \quad & (id_i, id_p) \in s.\mathsf{pmdb.known} \\ & \wedge \forall x \neq id_i.\ s.\mathsf{pmdb.serv}(x) \neq p2h(s, p) \\ & \wedge s.\mathsf{pmdb.serv}(id_i) \in \{p2h(s, p), \varepsilon\}. \end{aligned}$$

Now, if there exists a process p that wants to register a service, i. e. $\omega_p(s.\mathsf{pdb}(p)) = \text{REG } id_i\ id_p$, but one or more of the preconditions are not met, then an error message is returned to the caller. That is, the next SOS* state is computed by applying ERR to p's local state:

$$\begin{aligned} \Delta \supset \{\ & (s, res(s, p, \text{ERR})) \mid \\ & \exists id_i, id_p.\ \omega_p(s.\mathsf{pdb}(p)) = \text{REG } id_i\ id_p \wedge \neg vrega?(s, p, id_i, id_p)\}. \end{aligned}$$

If there exists a process p that wants to register a service and all preconditions are met, then it is ensured that p is registered for this services,

i. e. $s'.\mathsf{pmdb.serv}(id_i) = p2h(s,p)$ and $s'.\mathsf{pmdb.reg} = s.\mathsf{pmdb.reg} \cup \{(id_i, id_p)\}$. Furthermore, a success message is returned to the calling process.

$$\begin{aligned}\Delta \supset \{\ & (s, res(s', p, \text{SUCC})) \mid \\ & \exists id_i, id_p.\quad \omega_p(s.\mathsf{pdb}(p)) = \text{REG}\ id_i\ id_p \wedge vrega?(s, p, id_i, id_p) \\ & \qquad\qquad \wedge s' = s \begin{bmatrix} \mathsf{pmdb.serv}(id_i) := p2h(s,p), \\ \mathsf{pmdb.reg} := s.\mathsf{pmdb.reg} \cup \{(id_i, id_p)\} \end{bmatrix} \\ \}.\end{aligned}$$

Lookup a Service. The SOS implementation provides a system call that allows user applications to lookup the handle of the server providing a certain service. In SOS*, this call is represented by LUP id_i id_p, where id_i and id_p are the interface- and procedure id, respectively.

If there exists a process p that wants to lookup a service, i. e. $\omega_p(s.\mathsf{pdb}(p)) =$ LUP id_i id_p, but the service (id_i, id_p) is not registered (either because the service does not exist at all or because the service is not yet registered), then an error message is returned:

$$\begin{aligned}\Delta \supset \{\ & (s, res(s, p, \text{ERR})) \mid \\ & \exists id_i, id_p.\ \omega_p(s.\mathsf{pdb}(p)) = \text{LUP}\ id_i\ id_p \wedge (id_i, id_p) \notin s.\mathsf{pmdb.reg}\}.\end{aligned}$$

If there exists a process p that wants to lookup a service and there exists a process p_s that previously registered for this service, then a success message, including the handle to the providing server, is returned. That is, the handle data base is updated to reflect p's right to communicate with p_s, i. e. $s'.\mathsf{kds.hdb}(p)(hn_s) = p_s$. Furthermore, SUCC-LUP hn_s is applied to p's local state:

$$\begin{aligned}\Delta \supset \{\ & (s, res(s', p, \text{SUCC-LUP}\ hn_s)) \mid \\ & \exists id_i, id_p, p_s.\quad \omega_p(s.\mathsf{pdb}(p)) = \text{LUP}\ id_i\ id_p \wedge (id_i, id_p) \in s.\mathsf{pmdb.reg} \\ & \qquad\qquad \wedge p_s = h2p(s, s.\mathsf{pmdb.serv}(id_i)) \\ & \qquad\qquad \wedge hn_s = \begin{cases} min\{x \mid ph2p(s,p,x) = \varepsilon\} & \textbf{if } pp2h(s,p,p_s) = \varepsilon \\ pp2h(s,p,p_s) & \textbf{otherwise} \end{cases} \\ & \qquad\qquad \wedge s' = s[s.\mathsf{kds.hdb}(p)(hn_s) := p_s] \\ \}.\end{aligned}$$

Note, because we require $|pid\_t| \leq |hn\_t|$, assigning handles for, so far, unknown processes never fails. Because of this, we can be sure that either there exists already a mapping that points to the PID of the service provider, i. e. $pp2h(s,p,p_s)$, or there exists a (so far) unused handle, i. e. $\exists x.\ ph2p(s,p,x) = \varepsilon$, that may be used to establish a new mapping.

Unregister a Service. The SOS implementation provides a system call that allows a user applications to unregister a service it provides. In SOS*, this call is represented by UNREG $id_i\ id_p$, where id_i and id_p are the interface- and procedure id, respectively.

In order to successfully unregister a service, two preconditions must be met. The predicate *vunrega?* is satisfied, if these preconditions are met. For $\mathit{vunrega?}(s, p, id_i, id_p)$ to hold, p must be the process that serves the interface id_i and (id_i, id_p) must be registered. That is:

$$\mathit{vunrega?} \in S \times \mathit{pid\_t} \times \mathit{iid\_t}_\perp \times \mathit{prcid\_t}_\perp \rightarrow \mathit{bool}$$

$$\mathit{vunrega?}(s, p, id_i, id_p) \equiv s.\mathsf{pmdb.serv}(id_i) = \mathit{p2h}(s, p) \wedge (id_i, id_p) \in s.\mathsf{pmdb.reg}.$$

If there exists a process p that wants to unregister a service, i. e. $\omega_p(s.\mathsf{pdb}(p)) =$ UNREG $id_i\ id_p$, but one or more of the preconditions are not met, then an error message is returned to the caller:

$$\begin{aligned}\Delta \supset \{\ & (s, \mathit{res}(s, p, \text{ERR})) \mid \\ & \quad \exists id_i, id_p.\ \omega_p(s.\mathsf{pdb}(p)) = \text{UNREG}\ id_i\ id_p \wedge \neg \mathit{vunrega?}(s, p, id_i, id_p)\}.\end{aligned}$$

If there exists a process p (serving the interface id_s) that wants to unregister one of the services it is registered for, then this service is removed from the set of registered services, i. e. $s'.\mathsf{pmdb.reg} = s.\mathsf{pmdb.reg} \setminus \{(id_i, id_p)\}$, and a success message returned:

$$\begin{aligned}\Delta \supset \{\ & (s, \mathit{res}(s', p, \text{SUCC})) \mid \\ & \quad \exists id_i, id_p. \quad \omega_p(s.\mathsf{pdb}(p)) = \text{UNREG}\ id_i\ id_p \wedge \mathit{vunrega?}(s, p, id_i, id_p) \\ & \quad\quad \wedge \exists x \neq id_p.\ (id_i, x) \in s.\mathsf{pmdb.reg} \\ & \quad\quad \wedge s' = s[\mathsf{pmdb.reg} := s.\mathsf{pmdb.reg} \setminus \{(id_i, id_p)\}] \\ & \}.\end{aligned}$$

If there exists a process p (serving the interface id_s) that wants to unregister the last service it is registered for, then this service is removed from the set of registered services. Furthermore, unlike in the previous case, p is removed from the mapping indicating it as server providing services from the interface id_i, i. e. $s'.\mathsf{pmdb.serv}(id_i) = \varepsilon$. Finally a success message is returned:

$$\begin{aligned}\Delta \supset \{\ & (s, \mathit{res}(s', p, \text{SUCC})) \mid \\ & \quad \exists id_i, id_p. \quad \omega_p(s.\mathsf{pdb}(p)) = \text{UNREG}\ id_i\ id_p \wedge \mathit{vunrega?}(s, p, id_i, id_p) \\ & \quad\quad \wedge \nexists x \neq id_p.\ (id_i, x) \in s.\mathsf{pmdb.reg} \\ & \quad\quad \wedge s' = s\left[\begin{array}{l}\mathsf{pmdb.reg} := s.\mathsf{pmdb.reg} \setminus \{(id_i, id_p)\}, \\ \mathsf{pmdb.serv}(id_i) := \varepsilon\end{array}\right] \\ & \}.\end{aligned}$$

Undefined SOS Calls. It may be, that a user application calls the SOS but the call is unknown to the SOS. In such a case the SOS does not reply to the caller. In SOS* such calls are represented by UNDEF-SC. If there exists a process p, whose output is UNDEF-SC, then handling this call does not change the (visible) state:

$$\Delta \supset \{(s, s) \mid \exists p.\ \omega_p(s.\mathsf{pdb}(p)) = \text{UNDEF-SC}\}.$$

Undefined Kernel Calls. Similarly to an undefined SOS call, it could also happen that a user application calls the kernel but the call is unknown to the kernel. In such a case the kernel returns an error message. In SOS* such calls are represented by UNDEF-KC. If there exists a process p, whose output is UNDEF-KC, then handling this call results in an error message:

$$\Delta \supset \{(s, \mathit{res}(s, p, \text{ERR})) \mid \exists p.\ \omega_p(s.\mathsf{pdb}(p)) = \text{UNDEF-KC}\}.$$

IPC- Send and Receive. The kernel implementation provides system calls that allow user applications to communicate between each other. That is, user applications may send and receive messages. In SOS*, the send call is represented by SND $h_r\ m\ t_s$, where h_r is the handle of the receiving application, m is the message to send, and t_s is the send timeout. Analogously, the receive call is represented by RCV $h_s\ b\ t_r$, where h_s is the handle of the sending application, b is the buffer where to place the message, and t_r is the receive timeout. It may be that an application wants to receive a message from any application (other than a particular one). Such an open-receive call differs from a regular receive call in the specified handle. If $h_s =$ HN-NONE, then the calling application wants to do an open receive. The opposite operation, i. e. sending a broadcast, is currently not supported in SOS.

In order to successfully send or receive messages, the arguments supplied with the calls must match the required types. That means, for a (potentially) successful send operation the receiver handle h_r must point to a process known to the caller, i. e. $h_r \notin \{\varepsilon, \text{HN-NONE}\} \wedge \mathit{ph2p}(s, p, h_r) \neq \varepsilon$, and the message must be well formed, i. e. $m \neq \varepsilon$:[3]

$$\mathit{vsnda?} \in S \times \mathit{pid\_t} \times \mathit{hn\_t}_\perp \times \mathit{byte\_t}^* \cup \{\varepsilon\} \rightarrow \mathit{bool}$$

$$\mathit{vsnda?}(s, p_s, h_r, m) \equiv h_r \notin \{\varepsilon, \text{HN-NONE}\} \wedge \mathit{ph2p}(s, p_s, h_r) \neq \varepsilon \wedge m \neq \varepsilon.$$

Similarly, for a (potentially) successful receive operation the sender handle h_s must be HN-NONE or it must point to a process known to the caller, i. e. $h_s \neq \varepsilon \wedge \mathit{ph2p}(s, p, h_s) \neq \varepsilon$, and the buffer must be well formed, i. e. $b \neq \varepsilon$:[4]

$$\mathit{vrcva?} \in S \times \mathit{pid\_t} \times \mathit{hn\_t}_\perp \times \mathbb{N}_{32} \rightarrow \mathit{bool}$$

$$\mathit{vrcva?}(s, p_r, h_s, b) \equiv (h_s = \text{HN-NONE} \vee (h_s \neq \varepsilon \wedge \mathit{ph2p}(s, p, h_s) \neq \varepsilon)) \wedge b \neq \varepsilon.$$

[3]The well-formedness of a messages also includes things like memory safety. That means, if, for example, a message is not entirely in the memory region of the calling process, then, in SOS*, this message is represented by ε.

[4]Just like for well-formedness of messages, the well-formedness of buffers also include things like memory safety. Again, if, for example, a buffer is not entirely in the memory region of the calling process, then, in SOS*, this buffer is represented by ε.

Now, if there exists a process p_s that wants to send a message, i. e. $\omega_p(s.\mathsf{pdb}(p_s)) = \text{SND}\ h_r\ m\ t_s$, but one or more of the preconditions are not met, then an error message is returned to the caller:

$$\Delta \supset \{\ (s, \mathit{res}(s, p_s, \text{ERR}))\ |\ \exists h_r, m, t_s.\ \omega_p(s.\mathsf{pdb}(p_s)) = \text{SND}\ h_r\ m\ t_s \wedge \neg \mathit{vsnda?}(s, p_s, h_r, m)\}.$$

If there exists a process p_r that wants to receive a message, i. e. $\omega_p(s.\mathsf{pdb}(p_r)) = \text{RCV}\ h_s\ b\ t_r$, but one or more of the preconditions are not met, then an error message is returned to the caller:

$$\Delta \supset \{\ (s, \mathit{res}(s, p_r, \text{ERR}))\ |\ \exists h_s, b, t_r.\ \omega_p(s.\mathsf{pdb}(p_r)) = \text{RCV}\ h_s\ b\ t_r \wedge \neg \mathit{vrcva?}(s, p_r, h_s, b)\}.$$

In our implementation, messages are synchronously exchanged. That is, a pending send operation is completed by a matching receive operation and vice versa. Whether there exists a rendezvous situations is inspected by the predicate $\mathit{rv?}$. $\mathit{rv?}(s, p_s, h_r, p_r, h_s)$ is satisfied, if both parties want to communicate with each other. That is, each of the specified handles matches the PID of the opposite site or the handle specified by the sender matches the PID of the receiver and the receiver uses an open receive call ($h_s = \text{HN-NONE}$):

$$\mathit{rv?} \in S \times \mathit{pid\_t} \times \mathit{hn\_t} \times \mathit{pid\_t} \times \mathit{hn\_t} \rightarrow \mathit{bool}$$

$$\mathit{rv?}(s, p_s, h_r, p_r, h_s) \equiv p_r = \mathit{ph2p}(s, p_s, h_r) \wedge (p_s = \mathit{ph2p}(s, p_r, h_s) \vee h_s = \text{HN-NONE}).$$

If there exists a process p_r that wants to receive a message with a finite timeout, then it may be that this operation fails because of a timeout. That is, if the caller satisfied the preconditions $\mathit{vrcva?}$ but, up to now, the call could not be answered and currently there is no rendezvous situation, then the receive operation fails, returning a timeout error. This does not mean that any receive call with a finite timeout fails, but, unless an infinite timeout was specified, it could always be that such a call returns because of a timeout:

$$\begin{aligned}\Delta \supset \{\ & (s, \mathit{res}(s, p_r, \text{TIMEOUT}))\ | \\ & \exists h_s, b.\quad \omega_p(s.\mathsf{pdb}(p_r)) = \text{RCV}\ h_s\ b\ \text{FINITE} \\ & \qquad\quad \wedge\ \mathit{vrcva?}(s, p_r, h_s, b) \\ & \qquad\quad \wedge\ (\forall p_s, h_r, m, t_s.\ \omega_p(s.\mathsf{pdb}(p_s)) = \text{SND}\ h_r\ m\ t_s \wedge \neg \mathit{rv?}(s, p_s, h_r, p_r, h_s))\}.\end{aligned}$$

Similarly, if there exists a process p_s that wants to send a message with a finite timeout, then it may be that this operation fails because of a timeout:

$$\begin{aligned}\Delta \supset \{\ & (s, \mathit{res}(s, p_s, \text{TIMEOUT}))\ | \\ & \exists h_r, m.\quad \omega_p(s.\mathsf{pdb}(p_s)) = \text{SND}\ h_r\ m\ \text{FINITE} \\ & \qquad\quad \wedge\ \mathit{vsnda?}(s, p_s, h_r, m) \\ & \qquad\quad \wedge\ (\forall p_r, h_s, b, t_r.\ \omega_p(s.\mathsf{pdb}(p_r)) = \text{RCV}\ h_s\ b\ t_r \wedge \neg \mathit{rv?}(s, p_s, h_r, p_r, h_s))\}.\end{aligned}$$

If all goes well and there is a rendezvous situation, it may still be that the buffer b, provided by the receiver, is too small to fit the message m. In this case, the senders call fails with an error message. The receivers call, however, remains pending — waiting for a timeout or another matching send operation:

$$\begin{aligned}\Delta \supset \{\ & (s, \mathit{res}(s, p_s, \text{ERR})) \mid \\ & \exists p_r, h_s, h_r, b, m, t_s, t_r. \\ & \quad \omega_p(s.\mathsf{pdb}(p_s)) = \text{SND}\ h_r\ m\ t_s \land \omega_p(s.\mathsf{pdb}(p_r)) = \text{RCV}\ h_s\ b\ t_r \\ & \quad \land \mathit{vsnda?}(s, p_s, h_r, m) \land \mathit{vrcva?}(s, p_r, h_s, b) \\ & \quad \land \mathit{rv?}(s, p_s, h_r, p_r, h_s) \\ & \quad \land \mathit{len}(m) > b\}.\end{aligned}$$

Finally, if all goes well, there is a rendezvous situation, and the message fits into the receive buffer, then both parties receive a success message. That is, SUCC is applied to the senders (local) state and SUCC-RCV h_n m is applied to the receivers (local) state. The latter includes the actual message m and the (possibly new) handle h_n for the sender.

$$\begin{aligned}\Delta \supset \{\ & (s, \mathit{res}(\mathit{res}(s', p_r, \text{SUCC-RCV}\ h_n\ m), p_s, \text{SUCC})) \mid \\ & \exists h_s, h_r, b, t_s, t_r. \\ & \quad \omega_p(s.\mathsf{pdb}(p_s)) = \text{SND}\ h_r\ m\ t_s \land \omega_p(s.\mathsf{pdb}(p_r)) = \text{RCV}\ h_s\ b\ t_r \\ & \quad \land \mathit{vsnda?}(s, p_s, h_r, m) \land \mathit{vrcva?}(s, p_r, h_s, b) \\ & \quad \land \mathit{rv?}(s, p_s, h_r, p_r, h_s) \\ & \quad \land \mathit{len}(m) \leq b \\ & \quad \land \mathit{hn}_n = \begin{cases} \mathit{min}\{x \mid \mathit{ph2p}(s, p_r, x) = \varepsilon\} & \textbf{if } \mathit{pp2h}(s, p_r, p_s) = \varepsilon \\ \mathit{pp2h}(s, p_r, p_s) & \textbf{otherwise} \end{cases} \\ & \quad \land s' = s[s.\mathsf{kds}.\mathsf{hdb}(p_r)(\mathit{hn}_n) := p_s]\}.\end{aligned}$$

Note, this definition does not *explicitly* model the actual copy operation. This is because such a definition would very much depend on the particular process abstraction. However, just like ω_p computes the (generic) process output, we assume that δ_p updates the state according to the (generic) input. That means, for example, we assume $\mathit{res}(s', p_r, \text{SUCC-RCV}\ h_n\ m)$ updates the state of process p_r in such a way that the message m is copied to the place specified within the receive call (also see 5.2.4).

Local Computation. Last but not least, there are SOS* transitions that represent local computations of user applications. If there exists an application p, such that its output is equal to ε, then this application may do a local step:

$$\Delta \supset \{(s, s') \mid \exists p.\ \omega_p(s.\mathsf{pdb}(p)) = \varepsilon \land s' = s[\mathsf{pdb}(p) := \delta_p(s.\mathsf{pdb}(p), \varepsilon)]\}.$$

5.2.6 Runs

The model SOS* exhibits properties that can not be expressed *solely* by means of transition relation and state space. These properties are formalized by describing valid sequences of transitions, so-called runs.

We define a run to be an infinite sequence $r \in S^*$ such that $\forall n \in \mathbb{N}.\ (r^n, r^{n+1}) \in \Delta$. We define a (valid) SOS* run to be a run r such that $R(r)$, i. e. a sequence of states that is 'covered' by the transition relation, *and* that satisfies the predicate R (to be defined below).

Fairness between user applications is an important property. However, in our abstraction, the concrete scheduler is invisible. Thus, fairness can no longer be inferred by studying the scheduler and the interrupt handling mechanism. Here, we use runs to explicitly state this property.

Intuitively, fairness may be expressed by claiming that all applications eventually get to do something, and thereby change their state. In SOS* this is unfortunately not true as an application might wait infinitely long for another application to match its IPC operation. Thus, an application p might not progress, if it uses an infinite IPC call (in state s):

$$\begin{array}{c} \textit{possibly-no-progress} \in S_p \times \textit{pid\_t} \rightarrow \textit{bool} \\ \textit{possibly-no-progress}(s,p) \equiv \end{array}$$

$$\begin{array}{lll} (\exists h, m. & \omega_p(s.\mathsf{pdb}(p)) = \text{SND}\ h\ m & \text{INFINITE} \wedge \textit{vsnda?}(s,p,h,m)) \\ \vee\ (\exists h, b. & \omega_p(s.\mathsf{pdb}(p)) = \text{RCV}\ h\ b & \text{INFINITE} \wedge \textit{vrcva?}(s,p,h,b)). \end{array}$$

In order to formalize fairness between user applications, we also need to characterize progress. In SOS*, we can simply assume that a process p has progressed, between s and s', if s' could be the result of applying δ_p, with some input i, to p in s:

$$\begin{array}{c} \textit{progress} \in S_p \times \textit{pid\_t} \times S_p \rightarrow \textit{bool} \\ \textit{progress}(s,p,s') \equiv \exists i.\ s' = s[\mathsf{pdb}.(p) := \delta_p(s.\mathsf{pdb}(p), i)]. \end{array}$$

Being able to identify processes that might not progress and processes that have progressed, we can state fairness between user applications as predicates that must be satisfied by all valid SOS* runs.

The predicate *app-fairness-finite*(r) is satisfied, if each finite operation (of each process) is finally served.

$$\begin{array}{c} \textit{app-fairness-finite} \in S^* \rightarrow \textit{bool} \\ \textit{app-fairness-finite}(r) \equiv \\ \forall p, i.\ r^i.\mathsf{pdb}(p) \neq \varepsilon \wedge \neg\textit{possibly-no-progress}(r^i, p) \\ \Longrightarrow \\ \exists j \geq i.\ \textit{progress}(r^j, p, r^{j+1}). \end{array}$$

The predicate *app-fairness-infinite*(r) is satisfied, if each process that infinitely often *could* make progress infinitely often *does* make progresses.

$$\begin{array}{c} \textit{app-fairness-infinite} \in S^* \rightarrow \textit{bool} \\ \textit{app-fairness-infinite}(r) \equiv \\ (\forall p.\ (\forall i.\ \exists j \geq i, s'.\ r^j.\mathsf{pdb}(p) \neq \varepsilon \wedge \textit{progress}(r^j, p, s')) \\ \Longrightarrow \\ (\forall k.\ \exists l \geq k.\ \textit{progress}(r^l, p, r^{l+1}))). \end{array}$$

Now, R is simply the conjunction of all predicates defined over SOS* runs. For now, this is only *app-fairness-finite* and *app-fairness-infinite*. Thus:

$$R \in (S \times \Sigma)^* \rightarrow \mathit{bool}$$
$$R(r) \equiv \mathit{app\text{-}fairness\text{-}finite}(r) \wedge \mathit{app\text{-}fairness\text{-}infinite}(r).$$

5.3 Reasoning About Applications in SOS*

Non-determinism in a model is often a source of difficulty when it comes to verification. This is simply because it implies a larger search space than in a purely sequential model. In many cases non-deterministic models are still desirable as they allow to easily hide unnecessary details. In SOS*, for example, there are two such hidden details:

- The concrete scheduler is no longer visible. Instead, we chose non-deterministically which process to schedule next. This is commonly known as concurrency.
- There is no notion of time in SOS*. However, the implementation might cancel an IPC operation because of a timeout. Thus, in order to represent such situations, we terminate IPC operations non-deterministically and return an appropriate timeout error.

Now, when verifying a property of SOS*, we have to argue on all (fair) runs of the system. As this is a challenging task, we aim at reducing the search space. Similar to the reordering theory developed for reasoning about the concurrent execution of the processor and devices in Section 3.2, in this section we present a set of general theorems which reduce non-determinism caused by concurrency in SOS* and hence reduce the number of runs to analyze for correctness proofs.

5.3.1 Basic Observation

The basic observation for concurrency is that certain application steps may be reordered. For example, if a certain file remains unchanged, then the order of two concurrent read operations (on this file) is irrelevant in terms of the overall execution. Thus, we may arrange these non-interfering operations in any order we like. In general, such reordering is sound, if the steps do not interfere with each other. Applying reordering repeatedly, we can separate parts of execution traces of two applications, in which both are communicating only with each other. This may be useful, when verifying a communication protocol. During the RPC data transfer protocol, for example, neither the server nor the client communicates with other processes. Another argument for reordering is that complex operations may be grouped together to atomic steps. For example, a server may be considered to respond to requests immediately, while, in the implementation, there are many steps necessary to compute the response.

$$\begin{tikzcd}
& s_1 \arrow[dr, "ev_2"] & \\
s \arrow[ur, "ev_1"] \arrow[dr, "ev_2"'] & \Delta & s' \\
& s_2 \arrow[ur, "ev_1"'] &
\end{tikzcd}$$

Figure 5.2: Swapping Two Non-Interfering System Call Events

5.3.2 Non-Interference

For simplicity, in the remainder of this section, we assume that the mapping from handles to process ids is the identity. That means, we assume that the sets *pid_t* and *hn_t* are equal and that for all handles h and process ids p it holds:

$$ph2p(s, p, h) \neq \epsilon \implies ph2p(s, p, h) = h$$

On the one hand, this simplification does not affect the concept of rights associated with handles. Furthermore, the kernel still maintains the information on which process knows which other processes. On the other hand, an implementation that uses such a simple mapping function may expose some system properties. For example, the number of currently running processes or the uptime of the system could be inferred. One could extend the theory presented here to a general mapping from handles to PIDs; formulae just get slightly larger.

Before we can express non-interference, we need to introduce some notations.

In the following we write $s \xrightarrow{pid,o} s'$, if there is a valid SOS* transition from s to s' in which a system call $o \in \Omega_p$ of user process $pid \in pid\_t$ is processed.

$$\begin{aligned} s \xrightarrow{pid,o} s' \equiv \quad & \Delta(s, s') \\ & \wedge\, \omega_p(s.\mathsf{pdb}(pid)) = o \\ & \wedge\, \exists i.\ \delta_p(s.\mathsf{pdb}(pid), i) = s'.\mathsf{pdb}(pid). \end{aligned}$$

In case of an IPC-rendezvous situation, it may be that two (different) processes, i. e. the sender and the receiver, progress simultaneously. This is because, in the successful case, both of them receive a result message within a single SOS* step. This case cannot be expressed in the form $s \xrightarrow{pid,o} s'$. Thus, we introduce the more general idea of *system call events*. A system call event $ev \in pow(pid\_t \times \Omega'_p)$ is a set of at most two system calls:

$$s \xrightarrow{ev} s' \equiv \Delta(s, s') \wedge \forall (pid, o) \in ev.\ s \xrightarrow{pid,o} s'.$$

Now, we call two system call events ev_1, ev_2 *non-interfering*, denoted by $ev_1 \diamond ev_2$, if their execution can be swapped without modifying the result state (see

Fig. 5.2):

$$
\begin{aligned}
ev_1 \diamond ev_2 \equiv (\forall s, s'.\ \ & (\exists s_1.\ s \xrightarrow{ev_1} s_1 \wedge s_1 \xrightarrow{ev_2} s') \\
& \quad\quad\Leftrightarrow \\
& (\exists s_2.\ s \xrightarrow{ev_2} s_2 \wedge s_2 \xrightarrow{ev_1} s')).
\end{aligned}
$$

We extend the non-interference relation to sets of system call events:

$$Ev_1 \diamond Ev_2 \equiv \forall ev_1 \in Ev_1, ev_2 \in Ev_2.\ ev_1 \diamond ev_2,$$

and define the set of processes involved in an event by:

$$ap(ev) = \{p \mid \exists o.\ (p, o) \in ev\}.$$

Now, the following lemma defines the set of non-interfering system calls.

Lemma 36 (Non-Interfering System Calls) *Non-interfering system call events should not share involved processes: $ap(ev_1) \cap ap(ev_2) = \emptyset$. In the following we omit repeating this condition for each pair of system call events.*

Local transitions do neither interfere with local transitions nor with system calls of other processes:

$$ev \diamond \{(p, \epsilon)\}.$$

Portmapper lookup requests do not interfere with each other, but with portmapper register calls for the same procedure. Only portmapper register calls to different interfaces are non-interfering:

$$
\begin{gathered}
\{(p_1, \text{LUP } id_i\ id_p)\} \diamond \{(p_2, \text{LUP } id'_i\ id'_p)\} \\
(id_i = id'_i \implies id_p \neq id'_p) \implies \{(p_1, \text{LUP } id_i\ id_p)\} \diamond \{(p_2, \text{REG } id'_i\ id'_p)\} \\
id_i \neq id'_i \implies \{(p_1, \text{REG } id_i\ id_p)\} \diamond \{(p_2, \text{REG } id'_i\ id'_p)\}.
\end{gathered}
$$

IPC system call events are non-interfering, if they do not match with each other, i. e. they do not lead to a rendezvous situation:

$$
\begin{gathered}
(p_s, \text{SND } h_r\ m\ t_s) \in ev_1 \\
\wedge\ (p_r, \text{RCV } h_s\ b\ t_r) \in ev_2 \\
\wedge\ h_r \neq p_r \wedge h_s \neq p_s \\
\implies \\
ev_1 \diamond ev_2.
\end{gathered}
$$

The lemma above only defines the subset of the $\diamond$-relation relevant for the verification of RPC. Moreover, the $\diamond$-relation works solely on system call events and process identifiers, ignoring the current state. By extending the relation to states, one would, on the one hand, generate more complex non-interference assumptions, but, on the other hand, possibly identify more non-interfering actions. For example, in a file-system, a read operation and a write operation may be non-interfering due to missing rights of the writing process. These rights are not visible from the event but are encoded in the current state of the system.

5.3.3 Local Reasoning

Applications which do not invoke system calls at all during a computation — i. e. a finite run — can also be verified in isolation. That means we can, for instance, prove the correctness of such code, which neither involves communication with other processes nor with the SOS, sequentially in traditional Hoare logic. Note that the considered properties over the code can be either state- or termination properties, i. e. the lemmas can not be applied to simplify the verification of arbitrary temporal claims over the system.

For stating the lemma corresponding to the idea formulated above, we need some more notation. We denote the execution of i consecutive and local transitions of an application by δ_l:

$$\delta_l^i(x) = \begin{cases} x & \textbf{if } i = 0 \\ \delta_l^{i-1}(\delta_p(x, \epsilon)) & \textbf{otherwise.} \end{cases}$$

Furthermore, we call an execution of an application *isolated until step i*, if no system call is invoked:

$$\mathit{isolated?}(x, i) \equiv \forall k < i.\ \omega_p(\delta_l^k(x)) = \epsilon.$$

Now, we can generalize each isolated computation to an arbitrary one.

Lemma 37 (Isolated Computations) *The result of an isolated computation is neither influenced by the SOS nor by other processes:*

$$\mathit{isolated?}(x, i) \implies \forall r \in R, k \geq 0.\ r^k.\mathsf{pdb}(p) = x \implies \exists l \geq k.\ r^l.\mathsf{pdb}(p) = \delta_l^i(x)$$

The proof is based on the fairness of execution and on the fact that the SOS accesses an application only in response to a system call invoked by that application.

The following Lemma 38 gives us a tool to separate the verification of properties over two non-interfering sets of applications. In short, two sets of applications are non-interfering, if the system calls invoked by applications of the first set do not interfere with those invoked by applications of the second set.

First, we define, for a given run r and step number i, the set of system-call events, invoked between r^0 and r^i:

$$\mathit{out}(r, i) = \{\{(\mathit{pid}, o)\} \mid \exists j < i, \mathit{pid}.\ \omega_p(r^j.\mathsf{pdb}(\mathit{pid})) = o\}.$$

We define properties in a pre/post condition form. We formulate these properties over a restricted set of processes, i. e. we assume that the initial configuration only contains processes that are in a given set. Now, given some set X of process identifiers, a set Ev of system call events, and predicates P, Q, we call P, Q valid over X and Ev, if for all valid runs, with initial states satisfying P, we will finally reach a state satisfying Q. Furthermore, processes may only invoke system call events in Ev:

$$\begin{gathered} X, Ev \vdash P, Q \equiv \\ \forall r \in R.\ P(r^0) \wedge \forall q \notin X.\ r^0.\mathsf{pdb}(q) = \epsilon \implies \exists i.\ Q(r^i) \wedge out(r, i) \subseteq Ev. \end{gathered}$$

After the (above) preparatory work, we can now formulate a lemma which states that the correctness of computations, that do not interfere with each other, can be proven separately.

Lemma 38 (Separability of Non-Interfering Computations)

$$\begin{aligned} & X, Ev_x \vdash P_x, Q_x \\ \wedge\ & Y, Ev_y \vdash P_y, Q_y \\ \wedge\ & Ev_x \diamond Ev_y \\ & \implies \\ & (X \cup Y), (Ev_x \cup Ev_y) \vdash P_x, Q_x \\ \wedge\ & (X \cup Y), (Ev_x \cup Ev_y) \vdash P_y, Q_y \end{aligned}$$

The lemma above is extremely helpful for proving correctness of a communication protocol between two processes. Suppose the processes p and q only communicate with each other, i. e. they only invoke IPC send and IPC receive operations in which the sender and receiver handles are either p or q. Then we can generalize any correctness proof of this communication to arbitrary valid runs with arbitrary other processes. This is expressed in the following corollary, where the set $Ev_{p,q}$ denotes the above mentioned restriction to system call events:

Corollary 1 (Communication Protocols)

$$\forall X \subseteq pid\_t.\ \{p, q\}, Ev_{p,q} \vdash P, Q \implies \{p, q\} \cup X, \Omega_p \vdash P, Q.$$

5.3.4 Reordering

In the ideal case, reasoning about programs should be kept sequential and local as much as possible. Using Lemma 36, local steps of a process can always be grouped together. Furthermore, we can pretend that control is only switched in case of system calls.

This idea is expressed in the Reordering Theorem, claiming for each computation the existence of an equivalent and pure computation, in which local steps of different processes are (almost) never interleaved. Before stating the lemma, we first formalize the notion of *pure* computations. We say a computation is *p-pure* if control between processes is only switched due to a system call invocation or in case the currently executed process performs no more steps in the considered computation.

The latter case is expressed by the predicate *idle?*, formally defined over a range $[s, e]$ of step numbers in a given run r:

$$\mathit{idle?}(r, s, e, p) \equiv \forall s \leq i < e.\ r^i \xrightarrow{q,o} r^{i+1} \implies p \neq q.$$

Now, purity of a computation ranging from step s to step e can be formally defined by:

$$\begin{gathered} p\text{-}\mathit{pure?}(r, s, e) \equiv \\ \forall s \leq i < e.\ r^i \xrightarrow{p_1,o} r^{i+1} \xrightarrow{p_2,o'} r^{i+2} \wedge p_1 \neq p_2 \\ \implies \\ \omega_p(r^{i+1}.\mathsf{pdb}(p_1)) \neq \epsilon \vee \mathit{idle?}(r, i, e, p_1). \end{gathered}$$

Using the definition above our reordering theorem reads as follows:

Theorem 12 (Reordering) *For a given run r and step numbers s and e we can always find a corresponding pure computation on a run r_p evaluating to the same state:*

$$\forall r \in R, s, e.\ \exists r_p.\ p\text{-}\mathit{pure?}(r_p, s, e) \wedge r^e = r_p^e.$$

Proof Sketch The proof is done by induction on i. In the induction step we shift the local transition $i + 1$ to its appropriate block of isolated computations. This is done by a repeated application of swapping, which is possible for local transitions due to their non-interleaving nature.

The application of the reordering theorem leads to a significant reduction of the number of runs to analyze for proving a property over the whole system to be correct. Note that our reordering maintains liveness- and fairness properties, as always only a finite sequence is considered.

There is an intuitive way to think of pure computations: Consider system calls as function calls, where the functions are the processes from whose computation the result of the system call depends. For example, if process p invokes a receive from process q then we switch to the execution of process q until this one returns control again with a send call to p.

5.3.5 Summing Up

The presented theorems can be combined to the following proof strategy for communicating applications: By Lemma 38 we can restrict our analysis to runs where only applications of interest take steps. Using the Reordering Theorem we can assume that those runs are all *p-pure*, i.e. isolated computations of applications are grouped together. Finally, properties over those isolated chunks are verified according to Lemma 37 in traditional Hoare logic.

Chapter 6

Proving Correctness of Client/Server software

So far we have considered a model with concurrently executed processes running in the environment of the Simple Operating System. These processes communicate via IPC with each other and with the SOS.

Next, we extend the programming language C0 to system calls, instantiate them as applications in the SOS* and provide them with a more powerful communication mechanism than IPC: Remote Procedure Calls. RPC enables a process — the client — to invoke some service (i. e. a function) on a remote process — the server. From the client's point of view the invocation of the service should look similar to a local function call.

At compile time clients must know the names and the signatures of the services they intend to call. However, the location of the service (i. e. the name of the providing server) is then not necessarily known: During runtime the location of the service might change. Thus, we need a runtime mapping of service names to their locations. This mapping is provided by the SOS as described in Section 5.2: Servers register and clients look up services through invoking special SOS system calls.

After looking up the server handle, the client has to send the parameters of the call and, later on, receive the result. For each parameter and argument type we need to implement a pair of communication functions sending and receiving the data. For most types those functions can be implemented by single IPC calls. The only exception are pointer types, such as lists. Since IPC only supports copying data, which lies consecutively in the memory, for data structures that have to be traced over a chain of pointers, we need a more sophisticated mechanism.

Having these new communication functions, we define the protocol that the client must obey when requesting a service.

Next, we construct a simple example server in C0, identify the correctness criteria, and prove that it holds for our simple server. In short the correctness criteria states that if the client strictly obeys the protocol it will eventually receive the result of the invoked call. The correctness statement and proof for the example server can be easily transfered and generalized to other server implementations.

6.1 The Programming Language C0

In this section we instantiate the generic applications in the SOS* model to C0 applications. Since applications are interleaved, only small step semantics is appropriate. Thus, we have to extend the small step semantics of C0 to interactions with SOS system calls. Note, that in the following, we use monolithic C0 configurations, referring to them simply as C0 configurations. Moreover, we abbreviate access to the content of a variable evaluation $eval(c.\mathsf{conf}, \mathsf{VarAcc}(x)).\mathsf{content}$ by $va(c, x)$.

Our RPC mechanism is implemented as a set of *libraries* for C0. A library consists of a collection of function descriptors and a type environment, i.e. it has type $lib\_t = proctable_T \times tenv_T$. Given a library and a C0 configuration we can define, a predicate $isLinked? \in C_{\mathrm{com}} \times lib\_t \to \mathbb{B}$ which indicates if the program was linked to the given library. A simplified definition of the predicate is given by: $isLinked?(c, lib) \equiv fst(lib) \subseteq c.\mathsf{pst} \wedge snd(lib) \subseteq c.\mathsf{tenv}$.[1]

6.1.1 Semantics

The state space is instantiated to $C_{\mathrm{com}} \subset S_p$. Furthermore, we define the transition function $\delta^S_{c0} \subset \delta_p$ and the output function $\omega_{c0} \subset \omega_p$.

Transition function

The type of the transition function δ^S_{c0} is:

$$\delta^S_{c0} \in C_{\mathrm{com}} \times \Sigma_p \to C_{\mathrm{com}}.$$

The local part of this transition function is equivalent to ordinary C0 small step semantics, i. e. $\delta^S_{c0}(c, \epsilon) = \delta_{c0}(c.\mathsf{conf}, c.tenv, c.pt)$. The non-local parts involve invocations of system calls. In the program rest system call invocations appear as ordinary function calls. However, they are not part of the procedure table. Rather, the semantical effects of system calls on a C0 application is specified as a single atomic step in δ^S_{c0}, consuming the response of the system. For example, a portmapper lookup call has the following effect on a C0 application:

$$\begin{array}{ll} & c.\mathsf{conf}.\mathsf{prog} = x = \texttt{sc\_pm\_lkp}(iid, prcid) \\ \wedge & \delta^S_{c0}(c, \text{SUCC-LUP } h_s) = c' \\ \Longrightarrow & va(c', x)(0) = h_s. \end{array}$$

[1]Indeed linking is a much more complicated process, as types and global variables have to be defined unambiguously. For more informations refer to [IdR09].

IPC send and IPC receive system calls are slightly more complicated. Since C0 forbids type casts, we have to implement a pair of IPC functions for each type of data that should be sent or received. For simple types, IPC functions can be easily implemented and they can be assumed by the C0 programmer. Simple types cover all C0 types, in which no nested pointer types occur. The type restriction results from the fact that the basic IPC mechanism provided by the kernel only supports copying consecutive data chunks. Hence, types requiring pointer chasing can not be interpreted at this level.

Output function

If the head statement of the program rest is not a system call, the output function ω_{c0} returns ϵ. Otherwise, a corresponding output message is constructed. For example, a configuration c with the following head:

$$hd(s2l(c.\mathsf{conf}.\mathsf{prog})) = x = \texttt{sc\_ipc\_send\_int}(h_{exp}, m_{exp}, to_{exp})$$

generates the output:

$$\omega_{c0}(c) = \text{SND } t\text{-}hn(va(c, h_{exp}))\ t\text{-}m(va(c, m_{exp}))\ t\text{-}to(va(c, to_{exp})).$$

The functions $t\text{-}hn, t\text{-}m$ and $t\text{-}to$ map C0 values (i. e. content functions) to their corresponding interpretation in the SOS* abstraction. For example $t\text{-}to(va(c, t_{exp}))$ maps any timeout value which is not equal to the integer constant INFINITE to the constant FINITE. In the following, for simplicity, we will omit mentioning these translations, i. e. we write $va(\dots)$ instead of $t\text{-}x(va(\dots))$.

6.1.2 Doubly-Linked Lists in C0

Lists are provided to the C0 programmer by means of a generic library. For a given type T this library includes a type defining a doubly-linked list over elements of T. Furthermore, operations on lists are provided, such as creation, and element insertion and deletion. These operations have been formally verified against abstract lists [Sta06] (which are basically sequences on T).

6.2 Signatures of Services

The portmapper, maintained in the SOS, is not aware of the services' signatures. However, at compile time, the client must know the signatures and names of the services it intends to call. The signature of a service is given by the type of the input parameter and the type of the result:[2]

$$service\_sig\_t = (\mathsf{arg} : ty_{\mathrm{rpc}}, \mathsf{res} : ty_{\mathrm{rpc}}).$$

[2]In the context of RPC, the formalism describing the service signatures is called Interface Definition Language (IDL).

Here, ty_{rpc} denotes a subset of all possible C0 type descriptors. The formal definition of this subset is given in [Sha06]. In short, ty_{rpc} contains all simple and structured types. Pointer types are only allowed as part of the pre-defined doubly-linked lists.

Services are organized in so-called interfaces. The signature of an interface consists of a name and a mapping from procedure ids to corresponding signatures:

$$itfc\_t = (\mathsf{iid} : iid\_t, \mathsf{procs} : prcid\_t \rightarrow service\_sig\_t_{\epsilon}).$$

In the remainder of this paper we use the terms interface signatures and interfaces interchangeably.

6.3 Portmapper Correctness

The next lemma is a property of the interaction of different portmapper calls. This lemma states that, after a server has registered a service, any client that is looking up this service will finally receive the correct handle. Thus, it expresses the programmer's point of view of interacting portmapper calls.

Lemma 39 (Interacting Portmapper Register- and Lookup Calls)

Suppose there exists a step l at which a server with PID p_s wants to register the service (id_i, id_p). Further suppose that the server will not try to unregister the service, nor any other process will try to register the same service at any step during the run:

$$\begin{aligned}
&\forall r \in R, id_i, id_p, p_s, l.\\
&\quad \omega_{c0}(r^l.\mathsf{pdb}(p_s)) = \text{REG } id_i \ id_p\\
&\quad \wedge (\forall k, p'_s.\\
&\qquad\quad \omega_{c0}(r^l.\mathsf{pdb}(p'_s)) = \text{REG } id_i \ id_p \Longrightarrow p'_s = p_s\\
&\qquad \wedge \neg\omega_{c0}(r^l.\mathsf{pdb}(p_s)) = \text{UNREG } id_i \ id_p)\\
&\quad \Longrightarrow
\end{aligned}$$

Then, finally, a step in the run r is reached, after which the following holds: whenever some process with pid p_c looks up the service, it will receive a success message with the handle to the server p_s.

$$\begin{aligned}
&(\exists k > l.\ \forall j_1 > k.\\
&\quad \omega_{c0}(r^{j_1}.\mathsf{pdb}(p_c)) = \text{LUP } id_i \ id_p\\
&\quad \Longrightarrow (\exists j_2 > j_1.\ \delta^S_{c0}(r^{j_2}.\mathsf{pdb}(p_c), \text{SUCC-LUP } pp2h(p_c, p_s)) = r^{j_2+1}.\mathsf{pdb}(p_c)))
\end{aligned}$$

Proof Sketch. By applying Lemma 38, we can pretend that the lookup call is executed immediately after the execution of the register call. We only have to ensure that no system calls interfering with any of both portmapper calls is executed meanwhile. This follows from the assumptions of Lemma 39.

6.4 Sending and Receiving Data Structures

As described before, the IPC mechanism, provided in VAMOS does not fulfill the requirements for RPC. Therefore, C0 applications are supported with a library containing the implementation of RPC send and RPC receive functions. In the following we call these functions, RPC primitives.

These RPC primitives depend on the types of the data structure to be transmitted. Hence, we need a library generator, that produces, for a given interface signature *itfc*, the C0 implementation of functions for sending and receiving RPC messages.

The implementation is simple. For the sending side, non-list data structures are sent via a single IPC operation. List data structures are chased and sent element by element (recursively) via RPC. The receiver reconstructs the list, by chaining these elements together. Termination of both, the sending part and the receiving part, follows from the well-formedness of the list implementation.

This mechanism of packing and unpacking is an implementation detail and should not be visible to programmers of RPC servers and especially not to programmers of clients. Therefore, later on, we provide the programmers with a set of properties over the execution of those primitives, abstracting from their concrete implementation.

6.4.1 The Interface Compiler

The interface compiler takes as input an interface signature and generates a C0 library that contains the implementation of the corresponding RPC functions.

The function *genRPCprim* abstracts the implementation of the SOS interface compiler, i. e. it represents the semantics of the interface compiler. Since we are only interested in properties over the primitives, both the concrete code implementing *genRPCprim* and the specification defining the output of *genRPCprim*, is omitted:

$$genRPCprim \in itfc\_t \to lib\_t$$

$$genRPCprim(itfc) \equiv librpc_{itfc},$$

where $librpc_{itfc}$ contains for each type $T \in \{p.\mathsf{arg}, p.\mathsf{res} | p \in range(itfc.\mathsf{procs})\}$ occurring in the signature of one of the procedures of *itfc*, two functions:

1. `RPCsend_T`(h_r, arg, to) This primitive sends the argument $arg \in T$ to the application with handle h_r. The value *to* specifies some timeout value.[3] The function returns a value of type *rpcerr_t*, which indicates if the send

[3]Note that in the implementation, in contrast to SOS*, arbitrary integer values (except the constant INFINITE) can be specified for a finite timeout.

operation was successful, a timeout occurred, or the specified handle is not valid:

$$rpcerr\_t = \{\text{SUCC}, \text{TIMEOUT}, \text{INVALID}\}.$$

Thus, formally we have the following requirement to the library:

$$\begin{array}{l} fst(librpc_{itfc})(\texttt{RPCsend\_T}) = \\ (\mathsf{params} = [(h_r, \mathsf{Int}), (arg, T), (to, \mathsf{Int})], \mathsf{ret\text{-}type} = \mathsf{Int}, \ldots) \end{array}$$

2. `RPCrecv_T`(h_s, to) This primitive is called to receive an argument of type T from the application with the handle h_s. This call returns a value, which is a structure of type $rpcrcv\_T\_t$ containing two components. The first component indicates if the operation was successful, and the second one holds the received data:

$$rpcrcv\_T\_t = (\mathsf{status} : rpcerr\_t, \mathsf{data} : T).$$

Thus, formally we have the following requirement to the library:

$$\begin{array}{l} fst(librpc_{itfc})(\texttt{RPCrecv\_T}) = \\ (\mathsf{params} = [(h_s, \mathsf{Int}), (to, \mathsf{Int})], \mathsf{ret\text{-}type} = \mathsf{Int}, \ldots) \end{array}$$

These RPC primitives hide the details of sending and receiving parameters and results. This, for example, allows us to extend RPC to invocation of procedures via the Internet, without changing the signature of the RPC primitives. Even the specification would remain the same.

In the following we write for a function which in the domain of the procedure table of a library `RPCsend_T` $\in dom(lib)$ simply `RPCsend_T` $\in librpc_{itfc}$.

6.4.2 Predicates Signaling RPC Primitives

In order to specify the behavior of RPC primitives, we need to introduce a number of auxiliary predicates and functions that are used to describe the current state in the execution of the RPC primitives.

The predicate $beforeS?(c, T, resv, h_r, arg, to)$ evaluates to true if:

- The head of the program rest of c is an RPC send primitive, where T is the type of the parameter arg to be transferred, h_r is the handle of the receiver, and to is some timeout value.
- The C0 machine c was linked with an RPC library of an interface containing the argument type T.
- The result of the primitive is assigned to the C0 variable $resv$.

Formally we get:

$$beforeS? \in C_{\mathrm{com}} \times ty_{\mathrm{rpc}} \times var_n \times hn\_t \times expr \times \mathbb{N} \rightarrow bool$$

$$beforeS?(c, T, resv, h_r, arg, to) \equiv$$

$$\begin{array}{ll} \exists & itfc \in itfc\_t, h_{expr}, to_{expr} \in expr. \\ & \texttt{RPCsend\_T} \in librpc_{itfc} \wedge isLinked?(c, librpc_{itfc}) \wedge \\ & hd(s2l(c.\mathsf{conf}.\mathsf{prog})) = \quad resv = \texttt{RPCsend\_T}(h_{expr}, arg, to_{expr}) \wedge \\ & va(c, h_{expr}) = h_r \wedge \\ & va(c, to_{expr}) = to. \end{array}$$

The predicate $beforeR?(c, T, resv, h_s, to)$ evaluates to true, if:

- The head of the program rest of c is a receive RPC primitive, where T is the type of the parameter to be received, h_s is the handle of the sender and to is the timeout.
- The result of the primitive is assigned to the C0 variable $resv$.

Formally we get:

$$beforeR? \in C_{\mathrm{com}} \times ty_{\mathrm{rpc}} \times var_n \times hn\_t \times \mathbb{N} \rightarrow bool$$

$$beforeR?(c, T, resv, h_s, to) \equiv$$

$$\begin{array}{ll} \exists & itfc \in itfc\_t, h_{expr}, to_{expr} \in expr. \\ & \texttt{RPCrecv\_T} \in librpc_{itfc} \wedge isLinked?(c, librpc_{itfc}) \wedge \\ & hd(s2l(c.\mathsf{conf}.\mathsf{prog})) = \quad resv = \texttt{RPCrecv\_T}(h_{expr}, to_{expr}) \wedge \\ & va(c, h_{expr}) = h_s \wedge \\ & va(c, to_{expr}) = to. \end{array}$$

For a given C0 machine c and a function name fn, the predicate $duringFC?$ indicates that c is currently executing a call of the function fn:

$$duringFC? \in C_{\mathrm{com}} \times fun_n \rightarrow bool.$$

A formal description of $duringFC?$ is omitted.

The function $finishedFC$ indicates the time when the execution of a function is finished. For a given run r, step i, process id pid and a function name fn, it returns the first step, after i, in which the execution of function fn has finished:

$$finishedFC \in R \times \mathbb{N} \times pid\_t \times fun_n \rightarrow \mathbb{N}$$

$$finishedFC(r, i, p, fn) \ =$$
$$min\{j \mid j \geq i.\ duringFC?(r^{j-1}.\mathsf{pdb}(p), fn) \wedge \neg duringFC?(r^{j}.\mathsf{pdb}(p), fn)\}$$

Finally, we need one predicate that compares two states of a single C0 machine. The predicate *changed?* (c,c',v) evaluates to true, if in state c a function was called and in c' the same function returned. Furthermore, the result of the function call evaluates to the value v:[4]

$$\begin{aligned} \mathit{changed?}(c, c', v) \equiv \exists x, \texttt{fn}. & \\ & (\mathit{hd}(\mathit{s2l}(c.\mathsf{conf}.\mathsf{prog})) = x = \texttt{fn}(\dots) \\ \wedge \quad & c'.\mathsf{prog} = \mathit{tl}(\mathit{s2l}((c.\mathsf{conf}.\mathsf{prog}))) \\ \wedge \quad & \mathit{va}(c', x) = v) \end{aligned}$$

6.4.3 Properties of RPC Primitives

Instead of defining a new model, the semantics of the RPC primitives is given through a small set of theorems, describing *relevant parts* of their behavior. *Relevant parts* includes all properties which are needed for proving correctness of programs that use the RPC libraries. Similar to the correctness statement of the portmapper calls, the following lemmas should provide the programmer's point of view of the primitives. Thus, they are to be understood as specifications of the interface compiler's output. Given the concrete definition of *genRPCprim*, they can be discharged. Lemma 40 states that RPC primitives always terminate if a finite timeout was specified. Lemma 41 states that if two applications try to communicate via RPC send or RPC receive, then, finally, either the message will be transferred correctly or a timeout occurs. The lemmas are not proved for a concrete interface compiler. Rather the proof methodology is outlined.

Lemma 40 (Termination of RPC Function Calls)

This lemma states that every invocation of either the RPC send or RPC receive function terminates (possibly unsuccessfully, for example with a timeout error), in case the timeout value is finite:

1. $\forall r \in R, i \in \mathbb{N}, \mathit{sn} \in \mathit{pid\_t}, \mathit{to} \in \mathbb{N}.$
   $\mathit{beforeS?}(r^i.\mathsf{pdb}(\mathit{sn}), T, \dots, \mathit{to}) \wedge \mathit{to} \neq \text{INFINITE} \Longrightarrow$
   $\exists j > i.\ j = \mathit{finishedFC}(r, i, \mathit{sn}, \texttt{RPCsend\_T})$
2. $\forall r \in R, i \in \mathbb{N}, \mathit{receiver} \in \mathit{pid\_t}, \mathit{to} \in \mathbb{N}.$
   $\mathit{beforeR?}(r^i.\mathsf{pdb}(\mathit{rc}), T, \dots, \mathit{to}) \wedge \mathit{to} \neq \text{INFINITE} \Longrightarrow$
   $\exists j > i.\ j = \mathit{finishedFC}(r, i, \mathit{rc}, \texttt{RPCrecv\_T}).$

Proof Method. The C0 statements appearing in the implementation of the send- and receive primitives are either ordinary or IPC communication statements with finite timeout. By the fairness property of runs, we can deduce that each statement is finally executed (since at each configuration the C0 machine can make progress). The only remaining (general) source of non-termination is a type T defining an infinite data structure to process. However, such a type can only be defined by using pointers — and the only pointer structures allowed are well-formed and finite lists.

[4]Only functions with no side-effects are considered.

Lemma 41 (Correctness of RPC Communication Primitives)

Suppose, there is a sender *sn* and a receiver *rc* which want to communicate with each other using RPC primitives. The applications with ids *sn* and *rc* invoke the RPC send and RPC receive functions at steps i and j, respectively. Furthermore, suppose, the type of the transferred parameter *arg* is T.

$$\begin{aligned}
&\forall r \in R, i, j \in \mathbb{N}, sn, rc \in pid\_t, T \in ty_{rpc}, \\
&\quad resv_s, resv_r \in var_n, arg_s \in expr, to_s, to_r \in \mathbb{N}. \\
&\qquad \mathit{beforeS?}(r^i.\mathsf{pdb}(sn), T, res_s, \mathit{pp2h}(r^i, sn, rc), arg_s, to_s) \wedge \\
&\qquad \mathit{beforeR?}(r^j.\mathsf{pdb}(rc), T, res_r, \mathit{pp2h}(r^j, rc, sn), to_r) \wedge \\
&\qquad \Longrightarrow
\end{aligned}$$

Then there exist steps i' and j' in the run, at which the send and receive function calls have returned.

$$\begin{aligned}
&(\exists i', j' \in \mathbb{N}. \\
&\ i' = \mathit{finishedFC}(r, i, sn, \texttt{RPCsend\_T}) \wedge \\
&\ j' = \mathit{finishedFC}(r, j, rc, \texttt{RPCrecv\_T}) \wedge
\end{aligned}$$

and if furthermore the following two conditions hold:

- If the sender is waiting for the receiver, then the receiver will not try to receive messages (via IPC) from the sender until it invokes RPC receive.:

$$\begin{aligned}
&(i < j \Rightarrow \forall k \in \mathbb{N}.\ i \leq k \leq j \Rightarrow \\
&\quad \omega_{c0}(r^k.\mathsf{pdb}(rc)) \neq \text{RCV}\ \mathit{pp2h}(r^k, rc, sn)\ \cdots \wedge \\
&\quad \omega_{c0}(r^k.\mathsf{pdb}(rc)) \neq \text{RCV HN\_NONE} \ldots) \wedge
\end{aligned}$$

- If the receiver is waiting for the sender, the sender will not try to send messages (via IPC) to the receiver until it invokes RPC send:

$$\begin{aligned}
&(j < i \Rightarrow \forall k \in \mathbb{N}.\ j \leq k \leq i \Rightarrow \\
&\quad \omega_{c0}(r^k.\mathsf{pdb}(sn)) \neq \text{SND}\ \mathit{pp2h}(r^k, s, r)\ \ldots) \Longrightarrow
\end{aligned}$$

Then either the receiver will have received the correct data (sent by the sender) and a success message is reported to sender and receiver, or, in case the timeout was not set to infinite, a timeout could have occurred:

$$\begin{aligned}
&(\mathit{changed?}(r^i.\mathsf{pdb}(sn), r^{i'}.\mathsf{pdb}(sn), \text{SUCC}) \wedge \\
&\quad \mathit{changed?}(r^j.\mathsf{pdb}(rc), r^{j'}.\mathsf{pdb}(rc), \\
&\qquad (\mathsf{status} = \text{SUCC}, \mathsf{data} = \mathit{va}(r^i.\mathsf{pdb}(sn), arg_s)))\ ) \vee
\end{aligned}$$

$$\begin{aligned}
&(\ to_s \neq \text{INFINITE} \Rightarrow \mathit{changed?}(r^i.\mathsf{pdb}(sn), r^{i'}.\mathsf{pdb}(s), \text{TIMEOUT}) \wedge \\
&\ to_r \neq \text{INFINITE} \Rightarrow \mathit{changed?}(r^i.\mathsf{pdb}(rc), r^{i'}.\mathsf{pdb}(r), \\
&\qquad\qquad [\mathsf{status} = \text{TIMEOUT}, \mathsf{data} = ?]))\ )
\end{aligned}$$

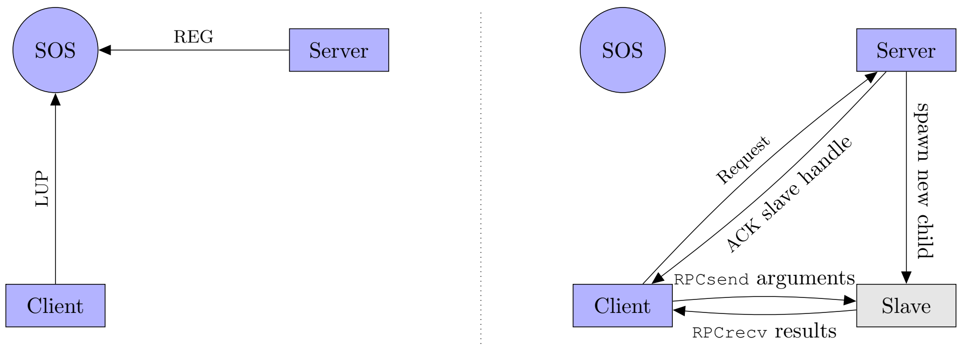


Figure 6.1: A sample execution of the RPC protocol — *left*: service register and lookup, *right*: server spawns child to handle client request

Proof Method. By using Lemma 38 we can prove the correctness of Lemma 41 ignoring all applications except the sender and the receiver. Applying the Reorder Theorem we can assume that control between sender and receiver is only switched at IPC-call borders. So, we can verify code segments between the invocation of two consecutive IPC operations purely sequentially in Hoare logic.

6.5 RPC Client Protocol

Having the new primitives of communication, we can define the protocol that the client must obey when requesting a service. Under a protocol we understand the sequence of messages that are sent and received by the client. This protocol should *not* depend on the concrete server architecture, but only on the service name. Figure 6.1 depicts the interaction of a client and a server.

The RPC client protocol is fairly simple. It consists of the following five steps:

- Request the location of the service via a portmapper call.
- Send the request to the server via IPC. This request contains the id of the desired (remote) procedure.
- Wait for an answer of the server. The answer contains a constant denoting whether the request was successful or not and the handle of the application processing the call (either the server itself or a newly spawned slave application dedicated for the request).
- Send, via RPC, the parameter of the service to the application referenced by the received handle.
- Wait, via RPC, for the result of the call.

6.5.1 CallService Library for C0 Applications

In order to release the programmer from manually implementing each step of the protocol, we provide him with a library compiler:

$$genCSprim \in itfc\_t \to lib\_t$$

$$genCSprim(itfc) = libService_{itfc}.$$

This compiler takes as input an interface signature and returns a C0 library containing the definition of the function `callService_iid_prcid`, for each service $(iid, prcid)$ in the given interface. Now, this function joins the individual steps of the above described client protocol into a single function call. The function `callService_iid_prcid` takes as parameters the argument of the service and a timeout value. Given the return type T of the service $(iid, prcid)$, `callService_iid_prcid` returns a value of type $csrcv\_T\_t$:

$$csrcv\_T\_t = (\mathsf{res} : cserr\_t, \mathsf{data} : T) \text{ , where}$$

$$cserr\_t = \{\text{SUCC}, \text{NOTREG}, \text{DENIED}\}.$$

6.5.2 Formal Description of the Client Protocol

For defining the RPC client protocol formally, we need three more predicates. The predicates *beforeCS?*, *request?* and *recvAnsw?* are used to characterize different aspects of the first three steps of the client protocol.

The predicate $beforeCS?(c, iid, prcid, arg, to, resv)$ evaluates to true if:

- The head of the program rest of c is the function call `callService_iid_prcid`, where $(iid, prcid)$ denotes the name of the service to invoke, arg denotes the argument to be passed to the server, and to is the timeout value used during communication with the server.
- The result of the function call is assigned to the C0 variable $resv$.

Thus, we get the definition:

$$beforeCS? \in C_{\mathrm{com}} \times iid\_t \times prcid\_t \times expr \times \mathbb{N} \times var_n \to \mathbb{B}$$

$$\begin{aligned}
&beforeCS?(c, iid, prcid, arg, to, resv) \equiv\\
&\exists itfc \in itfc\_t.\ itfc.\mathsf{iid} = iid \wedge prcid \in dom(itfc.\mathsf{procs}) \wedge\\
&isLinked?(c, librpc_{itfc}) \ \wedge \ isLinked?(c, libservice_{itfc})\\
&hd(s2l(c.\mathsf{conf}.\mathsf{prog})) = \quad resv = callService\_iid\_prcid(arg, to)
\end{aligned}$$

The predicate $request?(c, h_s, prcid)$ evaluates to true if:

- The C0 application c is sending (via IPC) a request to invoke procedure $prcid$ on the server with handle h_s.

- Directly after sending the request, the application will be waiting (via IPC) for an answer. The corresponding receiving buffer b must be large enough to store the answer of type integer (four bytes).

Thus, we get the definition:

$$request? \in C_{\text{com}} \times hn\_t \times prcid\_t \rightarrow bool$$

$$\begin{aligned}&request?(c, h_s, prcid) \equiv \exists\ msg \in byte\_t^*, b \geq 4.\\&\omega_{c0}(c) = \text{SND}\ h_s\ msg\ \text{INFINITE} \wedge\\&byte2prcid(msg) = prcid \wedge\\&\omega_{c0}(\delta_p(c, \text{SUCC})) = \text{RCV}\ h_s\ b\ \text{INFINITE}\end{aligned}$$

Here, the function $byte2prcid \in byte\_t^* \rightarrow prcid\_t$ translates a byte sequence to a procedure id.

The predicate $requestAnsw?(c, c', res, h_s)$ evaluates to true if:

- The C0 application in state c' has successfully returned from an IPC receive call in state c.
- The message res and the handle h_s were received. In case the IPC message was an RPC request, res should encode whether the request was acknowledged or not, i. e. its value is either ACK and NACK.ACK and NACK are constants of type $\mathbb{Z}_{32}$. If an ACK message was received, h_s denotes the handle of the application that will serve the request.

Thus, we get the definition:

$$requestAnsw? \in C_{\text{com}} \times C_{\text{com}} \times \mathbb{Z}_{32} \times hn\_t \rightarrow bool$$

$$\begin{aligned}&requestAnsw?(c, c', res, h_s) \equiv\\&\exists msg \in byte\_t^*.\ \delta_p(c, \text{SUCC-RCV}\ h_s\ msg) = c' \wedge\\&byte2ack(msg) = res.\end{aligned}$$

Here, the function $byte2ack \in byte\_t^* \rightarrow \{\text{ACK}, \text{NACK}\}$ translates a byte sequence to a server response.

After defining these auxiliary predicates we can now specify the intended behavior of the *callService* library. The code of any C0 library claiming to implement the client protocol, must fulfill Theorem 13.

In the following theorem we use the abbreviations: $T_{arg} = itfc.\mathsf{procs}(prcid).arg$ and $T_{res} = itfc.\mathsf{procs}(prcid).\mathsf{res}$.

Theorem 13 (Behavior of CallService)

$$\forall r \in R, i_0 \in \mathbb{N}, iid \in iid\_t, prcid \in prcid\_t, c \in pid\_t, to \in \mathbb{N}, arg \in expr.$$

1. Assume that the next statement to execute of the client process c is an invocation of a service (iid, $prcid$). Then, finally, the client will be requesting the location of the service from the SOS.

$$\begin{aligned}&\mathit{beforeCS?}(r^{i_0}.\mathsf{pdb}(c), iid, prcid, arg, \text{INFINITE}, res_{cs})\\ &\quad\Longrightarrow (\exists i_1 > i_0.\ \omega_{c0}(r^{i_1}.\mathsf{pdb}(c)) = \text{ LUP } iid\ prcid \wedge\end{aligned}$$

2. If the portmapping request was successful and the service location h_s was returned, the client will send an RPC request to the server:

$$\begin{aligned}&(\forall\ i_2.\ i_2 = min\{j > i_1 \mid progress(r^{j-1}, c, r^j)\} \wedge\\ &\delta^S_{c0}(r^{i_2-1}.\mathsf{pdb}(c), \text{SUCC-LUP } h_s) = r^{i_2}.\mathsf{pdb}(c)\\ &\quad\Longrightarrow (\exists i_3 > i_2.\ \mathit{request?}(r^{i_3}.\mathsf{pdb}(c), h_s, prcid) \wedge\end{aligned}$$

3. If the answer to the request was positive, the handle h_s of the process that will serve the call is received. The client will next try to send the arguments of the call via RPC to the process h_s:

$$\begin{aligned}&(\forall i_4 > i_3.\ i_4 = min\{j \mid \mathit{requestAnsw?}(r^{i_3}.\mathsf{pdb}(c), r^j.\mathsf{pdb}(c), b, h_s)\} \wedge b = \text{ACK}\\ &\quad\Longrightarrow (\exists i_5 > i_4, res_{send}.\ \mathit{beforeS?}(r^{i_5}.\mathsf{pdb}(c), T_{arg}, res_{send}, h_s, \text{INFINITE}) \wedge\end{aligned}$$

4. If the RPC sending of the argument was successful, the client will wait (via RPC) for the result of the call.

$$\begin{aligned}&(\forall i_6 > i_5.\ i_6 = \mathit{finishedFC}(r, i_6, c, \texttt{RPCsend\_T}_{arg}) \wedge\\ &\qquad va(r^{i_5}.\mathsf{pdb}(c), res_{send}) = \text{SUCC}\\ &\quad\Longrightarrow (\exists i_7 > i_6, res_{rcv}.\ \mathit{beforeR?}(r^{i_7}.\mathsf{pdb}(c), T_{res}, res_{rcv}, h_s, \text{INFINITE}) \wedge\end{aligned}$$

5. If the client successfully receives the result, then `callService_iid_prcid` will terminate and return the success message as well as the received data.

$$\begin{aligned}&(\forall i_8 > i_7.\ i_8 = \mathit{finishedFC}(r, i_7, c, \texttt{RPCrecv\_T}_{res})\\ &\qquad \wedge va(r^{i_8}.\mathsf{pdb}(c), res_{rcv}.\mathsf{res}) = \text{SUCC}\\ &\quad\Longrightarrow (\exists i_9 > i_8.\ i_9 = \mathit{finishedFC}(r, i_8, c, \texttt{callService\_iid\_prcid}) \wedge\\ &\qquad \mathit{changed?}(r^{i_0}.\mathsf{pdb}(c), r^{i_9}.\mathsf{pdb}(c),\\ &\qquad\qquad [\mathsf{res} = \text{SUCC}, \mathsf{data} = va(res_{rcv}.\mathsf{data})]))))))))))\end{aligned}$$

Proof Method. The lemma has not been proven for a concrete library generator. We rather outline the proof methodology, here. By applying Lemma 37 we can generalize properties proved locally to properties that also hold in the interleaved setting of SOS*. That means, the correctness of a given *callService* library can be proven almost completely sequentially (except when waiting for results from the SOS or other applications) and in traditional Hoare logic.

For example, for step 1 of Theorem 13, it suffices to prove the following, local property over the library code: $\mathit{beforeCS?}(c, iid, prcid, arg, \text{INFINITE}, res_{cs}) \Longrightarrow \exists i.\ \omega_{c0}(\delta^i_l(c)) = \text{ LUP } iid\ prcid$.

6.6 Building a Server and Proving its Correctness

Now we have everything we need for building arbitrary servers and specify their correctness: We can register services, send and receive messages of dynamic length and we know how a *correct* client should behave. In the next section, we give the implementation of a simple server and prove its correctness. Under correctness we understand, that whenever the client obeys the above described protocol, it will eventually receive the result of the invoked call.

Even so, one could implement the same interface of the Math Server through many different architectures — for instance, a simple architecture might handle one request after the other, whereas a more advanced architecture might spawn a new slave process for each request — all architectures have to fulfill the same correctness criteria. This criteria represents the client's view on the server.

6.6.1 An Example: Math Server

The *MathServer* implements the interface *mathitfc* with only one procedure. This procedure takes two integers and returns their product. Hence, the parameter type of the service is given through:

$$rpcmul\_t = (\mathsf{x} : \mathbb{Z}_{32}, \mathsf{y} : \mathbb{Z}_{32})$$

The interface id is MATH and the procedure id is MUL. *mathitfc* is formally described through:

$$mathitfc = [\mathsf{iid} = \text{MATH}, \mathsf{procs} = (\lambda x.\ \textbf{if}\ x = \text{MUL}\ \textbf{then}\ (\mathsf{arg} = rpcmul\_t, \mathsf{res} = \mathbb{Z}_{32}))]$$

If we program a server, we first have to link it with the three libraries Libvamos, Libsos and librpc. The library Libvamos provides us with the functions for sending requests and receiving answers via IPC: `sc_ipc_send_int` and `sc_ipc_rcv_rpcreq_t`.[5] The library Libsos provides us with the portmapping call `sc_pm_reg`, and $librpc_{mathitfc}$ contains the RPC send and RPC receive functions needed to transmit arguments and results of the MUL procedure: `RPCrecv_rpcmul_t` and `RPCsend_int`.

The implementation of *MathServer* is pretty simple. Listing 6.6.1 gives the code of the main function of *MathServer*. First the service is registered in the SOS. Then the main loop is entered. There, the server first waits for incoming requests via IPC. If a request to execute service MUL is received, the server sends an acknowledgment and waits for an RPC message containing the argument of the call. If, finally, the argument arrives, the result is calculated and sent to the client via RPC.

[5] Libvamos contains C0 macros, for generating IPC functions of the desired type.

```
dummy = sc_pm_reg (MATH, MUL);
while (true) {
  // open receive for service requests
  dummy = sc_ipc_rcv_rpcreq_t (HN_NONE, request, client, INFINITE);
  // dispatching
  if (*request.prcd == MUL) {
    // sending acknowledgement message
    dummy = sc_ipc_send_int (*client, ACK, INFINITE);
    // receiving arguments of the client with infinite timeout
    clientmsg = RPCrecv_rpcmul_t (*client, INFINITE);
    // computing the result of the requested service
    res = clientmsg.data.x * clientmsg.data.y;
    // sending the result to the client
    dummy = RPCsend_int (*client, res, INFINITE);
  }
  // in case of a call to a procedure other than MUL send denial
  else {
    dummy = sc_ipc_send_int (*client, NACK, 10);
  }
}
```

Figure 6.2: Implementation of Math Server — variables *client* and *request* are pointers to integers and * denotes the dereferencing operator.

6.6.2 Correctness of Math Server

The correctness theorem assumes that all applications requesting the MUL service obey the RPC client protocol. This condition seems to be too strict. The reason for that condition, is that, meanwhile the server waits for the arguments of a remote call, it will block — possibly infinitely long. Thus, it must *trust* that the client eventually will send or receive the data.

Alternatively, the server could specify a finite timeout when waiting for the client. This solution is cumbersome when it comes to verification: It could happen that the client, although obeying the protocol, is simply *tooslow* for sending the data. In practice one could easily try to estimate a robust timeout value for the server. However, proving that this value will never lead to a timeout is much harder, since it depends on the implementation of the hardware, the kernel, etc (for a detailed worst-case execution time analysis, covering a complete system stack, refer to [KP06]).

Yet, if we know that *all* clients during a run will obey the protocol, then we can set the timeout values of the server to infinite and hereby prove correctness *and* termination for all calls.

The above mentioned condition is formulated in the predicate *goodclients*. It is satisfied if, during the whole run, every client will only communicate with servers, in the context of a `callService` function call. With other words, it is true, if all clients communicating with the server obey the RPC client protocol.

$$goodClients? \in R \times iid\_t \times prcid\_t \rightarrow bool$$

$$goodClients?(r, iid, prcid) \equiv$$

$$\begin{aligned}
&\forall i \in \mathbb{N}, s, c \in \mathit{pid\_t}.\\
&\quad r^i.\mathit{pmdb.serv}(\mathit{iid}) = \mathit{pp2h}(r^i, \mathit{OSPID}, s) \wedge (\mathit{iid}, \mathit{prcid}) \in r^i.\mathsf{pmdb.reg} \wedge\\
&\quad \omega_{c0}(r^i.\mathsf{pdb}(c)) = \text{SND } \mathit{pp2h}(r^i, c, s) \ \ldots\\
&\qquad \Longrightarrow \mathit{duringFC?}(r^i.\mathsf{pdb}(c), \texttt{callService\_iid\_prcid})
\end{aligned}$$

Now, we can express the correctness of the our Math Server. It states, that if all clients obey the client protocol, finally the MUL service will be registered and any client requesting it will finally receive the correct answer:

Theorem 14 (Correctness of Math Server)

If, during a run, the *MathServer* application is started, all possible clients are *good* and no other application tries to register the interface *mathitfc*:

$$\begin{aligned}
&\forall r \in R,\ \ s, c \in \mathit{pid\_t}, i \in \mathbb{N}.\\
&\quad r^i.\mathsf{pdb}(s) = \mathit{MathServer}\ \wedge\ \mathit{goodClients?}(r, \text{MATH}, \text{MUL}) \wedge\\
&\quad (\forall k \in \mathbb{N}, s' \in \mathit{pid\_t}.\ \ \omega_{c0}(r^k.\mathsf{pdb}(s')) = \text{REG MATH ?} \Longrightarrow s' = s)\\
&\\
&\quad \Longrightarrow,
\end{aligned}$$

then a step k in the run is reached, after which the following holds: if some application calls the service (MATH, MUL) with integers x and y as parameters, then, finally, this call will return with the correct result, i. e. the product of x and y:

$$\begin{aligned}
&(\exists k > i.\ \forall j > k, \mathit{arg} \in \mathit{expr}.\\
&\quad \mathit{beforeCS?}(r^j.\mathsf{pdb}(c), \text{MATH}, \text{MUL}, \mathit{arg}, \text{INFINITE}, \ldots) \Longrightarrow\\
&\quad (\exists j' > j.\ j' = \mathit{finishedFC}(r, j, c, \mathit{callService}\_\text{MATH}\_\text{MUL}) \wedge\\
&\quad \mathit{changed?}(r^j.\mathsf{pdb}(c), r^{j'}.\mathsf{pdb}(c),\\
&\qquad [\mathsf{res} = \text{SUCC}, \mathsf{data} = \mathit{va}(r^j.\mathsf{pdb}(c), \mathit{arg}.\mathsf{x}) * \mathit{va}(r^j.\mathsf{pdb}(c), \mathit{arg}.\mathsf{y})])))
\end{aligned}$$

Proof. Since in SOS* a fair scheduling is assumed, eventually the first line in Listing 6.6.1 is executed. At that point all assumptions of Lemma 39 are satisfied and we conclude that finally (at step k) the service is registered in the portmapping component of the SOS. On the client side, it follows from Theorem 13 part 1 that the client will request the SOS for the address of the service:

$$\begin{aligned}
&\ldots\ \mathit{beforeCS?}(r^j.\mathsf{pdb}(c), \text{MATH}, \text{MUL}, \mathit{arg}, \text{INFINITE}, \ldots)\\
&\Longrightarrow \exists j_1 > j.\ \omega_{c0}(r^{j_1}.\mathsf{pdb}(c)) = \text{LUP MATH MUL}
\end{aligned}$$

From Lemma 39 follows that the service look up of the client will be successful, and from Theorem 13 part 2 follows that the client is finally sending a request to the server via IPC:

$$\begin{aligned}
&\ldots\ \mathit{beforeCS?}(r^j.\mathsf{pdb}(c), \text{MATH}, \text{MUL}, \mathit{arg}, \text{INFINITE}, \ldots)\\
&\Longrightarrow \exists j_2 > j.\ \mathit{request}(r^{j_2}.\mathsf{pdb}(c), s, \text{MUL})
\end{aligned}$$

Now, (at step j_2) the server can be in two different states. Either it is waiting in an open receive for requests (line 3), or the server is already processing some request (lines 5 - 18). In the second case we have to show, that the server will finally finish the processing and return to the beginning of the main loop. For that we only have to prove the termination of each statement of the loop.

Since the timeout values of the RPC send and receive statements are set to infinite, termination does not follow directly.[6] By using Lemma 41 and because we know that the client is obeying the client protocol, we prove that each RPC send primitive on server side is matched by an RPC receive primitive on client side (where the timeout is set to infinite) and conclude rendezvous and termination. Similarly we prove termination of the RPC receive primitive.

Hence, the server will always be finally waiting for requests. By the fairness property *app-fairness-infinite* over the run r we conclude that the server finally will also receive the request of client c. Since the requested procedure is MUL, the server will send an acknowledgment (line 7). From Theorem 13 part 3 it follows that finally the client will send the service parameters to the server:

$$\begin{aligned}&\ldots\ \mathit{beforeCS?}(r^j.\mathsf{pdb}(c), \mathrm{MATH}, \mathrm{MUL}, arg, \mathrm{INFINITE}, \ldots)\\ &\Longrightarrow \exists j_3 > j.\ \mathit{beforeS?}(r^{j_3}.\mathsf{pdb}(c), arg, ?, \mathit{pp2h}(r^{j_3}, c, s), \mathrm{INFINITE})\end{aligned}$$

Next, the server will try to receive the parameters via RPC (line 9). Using Lemma 41 one can easily prove that the parameters of the client are correctly transmitted to the server. Hence, from Theorem 13 part 4 it follows that, the client will wait, via RPC, for the result:

$$\begin{aligned}&\ldots\ \mathit{beforeCS?}(r^j.\mathsf{pdb}(c), \mathrm{MATH}, \mathrm{MUL}, arg, \mathrm{INFINITE}, \ldots)\\ &\Longrightarrow \exists j_4 > j.\ \mathit{beforeR?}(r^{j_4}.\mathsf{pdb}(c), \mathbb{Z}_{32}, ?, \mathit{pp2h}(r^{j_4}, c, s), \mathrm{INFINITE})\end{aligned}$$

On server side, next, the service (line 11), i. e. the computation of the product of the received integers, is executed. Since we assume the correctness of the C0 multiplication algorithm, it follows that the server will finally send via RPC the correctly computed product to the client (line 13). Using again Lemma 41 and Theorem 13 part 5 we conclude that the client will finally receive the result and return from the call with a success message:

$$\begin{aligned}&\ldots\ \mathit{beforeCS?}(r^j.\mathsf{pdb}(c), \mathrm{MATH}, \mathrm{MUL}, arg, \mathrm{INFINITE}, \ldots)\\ &\Longrightarrow \exists j' > j.\ j' = \mathit{finishedFC}(r, j_4, c, \text{callService\_MATH \_MUL}) \wedge\\ &\qquad\qquad \mathit{changed?}(r^j.\mathsf{pdb}(c), r^{j'}.\mathsf{pdb}(c'),\\ &\qquad\qquad\qquad [\mathsf{res} = \mathrm{SUCC}, \mathsf{data} = va(r^j.\mathsf{pdb}(c), arg).x * va(r^j.\mathsf{pdb}(c), arg).y])\end{aligned}$$

q.e.d

The proof technique presented here, obviously applies also to server implementations other than the simple Math Server discussed in this Section.

[6]In case timeouts were set to a finite value, termination would follow from Lemma 40.

Chapter 7

Conclusion and Future Work

The thesis contains to the best of our knowledge the first formal functional verification of a device driver (the hard disk driver) and for the first time the outline of a paper and pencil proof of a realistic client server mechanisms at the code level. It splits into these two main parts:

- Drivers Verification. First we extended a language stack — reaching from the gate-level implementation of a processor up to the high-level semantics of C0 — to deal with device drivers. This comprised, a reordering theory to sequentialize the interleaved computation of the processor and devices. The result is a methodology for driver verification in the context of pervasive system verification which is general enough to be applicable to other drivers in other settings, and concrete enough to expose 'real' verification problems and hurdles.

  Subsequently, the language stack has been leveraged to verify an important piece of kernel correctness: the hard disk driver used for swapping-out pages. During the verification process, we had to join theories, models and proofs of many people (cf. Section 7.1) and on many different semantical layers to finally obtain a single theorem covering the whole system. This theorem has been successfully used to verify the kernel. During driver verification we were forced to reason about many aspects and conditions of the system which are usually under the veil of 'technically but simple'. They don't show up until pervasive and formal verification is conducted. These conditions may be crucial for system correctness, as we illustrated for memory consumption (for which we verified a small theory to estimate it).

- Client/Server Verification. We used arguments and abstractions known in one formalism or another from previous work. As explained in the introduction, the contribution of this work is, however, to put all these concepts into a single mathematical theory which:

- *specifies the correctness* of systems as they are *without simplifications*, i. e. *at the code level*, and
- *justifies all abstractions* (used on the way) by proving that they are implemented correctly by lower system layers.

First of all, we presented a subset of the specification of Bogan's simple operating system that supports communicating user applications. Then, we showed how the correctness of code, running in this concurrent setting, can be proven (almost) sequentially in traditional Hoare logic. For this we identified classes of non interfering system calls. Finally, we presented the specification and outlined the verification of an RPC mechanism. We skipped over the not so simple theory of linking, required to show that functions linked to a C0 program behave — under certain conditions — in the desired way. Such a theory will be presented in [IdR09].

On top of our work, using the presented RPC mechanism, an email system, covering an SMTP server and an email client, has been constructed and partially verified. Thus, the work at hand is not only an abstract case-study, but has parially been applied to a concrete software system.

7.1 Formal Work

Integrating the huge amount of specifications, models and proofs emerged as a highly non-trivial and time-consuming engineering task. This covers, among other things, a social process, in which the work of many researchers, located at different places, has to be combined to one uniform and formal Isabelle/HOL theory. More than 250 Isabelle theories developed by more than 10 researchers were either directly or indircetly imported in our work (cf. Appendix A).

On the model side, we mechanized the VAMP assembly with devices model and the C0 with XCAll semantics. On the proof side, we mechanized among others the reordering theory, the theory on estimating memory consumption, theory for integrating devices into the C0 language stack (adapted for the use in the CVM verification) and the whole hard disk driver for swapping out pages (i.e. Theorem 11 was formally verified, where in the pervasive proof chain only correctness of the read case had been assumed without proof). We adapted this driver to be applicable in the context of formal page-fault handler verification. Theorem 11 has been successfully used in Isabelle/HOL to verify the CVM kernel.

All together, we have carried out almost 13.000 proof steps in 411 Lemmas (cf. Appendix B).

In the process of formal driver verification, we developed a series of *verification-engineering* tools to facilitate the proof development. Among others, these are:

- Clone-Detection. We applied ConQAT, a clone detection tool[1], to the Verisoft theory corpus and to the driver verification sources. For Verisoft, an analysis showed that up to 28% of the proof scripts (larger than 10 lines) are duplicated. For the driver verification, we had about 23% duplicates. Sometimes, these duplicates are justified, however in many cases they are not. In general, there are two classes of unnecessary duplicates: First, two lemmas with different statements could still have the same proof script. The identical proof may be an indication of a more general lemma subsuming the other two as sub cases. Second, some proof scripts are repeated within larger proofs of many different lemmas. These scripts may, for example prove some simple arithmetic equality which is used at different locations. In such cases, one should refactor these proof lines into a separate lemma.

- Subgoal-Detection. Similarly, we have implemented a tool to detect recurring conclusions of subgoals (across many proofs). In an online analysis, the prover can warn you if you are about to establish a conclusion that you have already proven earlier (not implemented yet). In an offline analysis, recurring conclusions of subgoals can just be reported by number of recurrences; the higher the number of repetitions, the higher the potential merit in factoring out the conclusion into a separate lemma. In this case, the prover can also suggest potential assumptions for the lemma, e. g. the intersection of the sets of assumptions that have appeared for that particular conclusion. An initial analysis for the C0 small-steps semantics (which contains many lemmas, e. g. on type safety) has shown that ca. eight percent of the subgoal conclusions appear repeatedly. This number should be treated with care, however, because if a certain chain of conclusions appears multiple times (because the same proof is conducted multiple times), than all conclusions in all instances of the chain are counted multiple times.

7.2 Future Work

There are several directions of possible future work:

- Driver verification. The driver for page swap-in remains to be verified. Also, the driver given is only a polling one. For the file system implementation in Verisoft's simple operating system interrupts are also used.

  For the verification of code for devices other than a hard disk, it might be interesting to refine the concept of stability, which was introduced in Sect. 3.2. Typically, a communicating device (e.g., network interface

[1] http://conqat.in.tum.de

card) is never stable on the complete configuration because it always asynchronously transfers data. However, by defining stability only for parts of the state, it can still usually be preserved. For example, communication is often channeled through buffers for transmission and reception with processor and environment accessing these buffers at different ends. Concurrency can be reduced in such a scenario.

- Client/Server verification. The formal verification in Isabelle/HOL of the reported RPC results is still an open task. The problems and challenges related to this process, are not specific to the verification of RPC, but has been experienced and to a large extend solved in the formal verification of the driver framework.

- Proof Engineering. Larger efforts should be undertaken to simplify and better organize the formal verification in a computer aided proof system as Isabelle/HOL. On the one hand side it would be desirable to have more such analysis tools as the one described above. We think, that in projects with such a large theory corpus, 'proof-by-search' technology (as Subgoal-Detection) may be highly promising.

  On the other hand, the use of automatic tools in Verisoft often failed due to a huge overhead caused by the integration of external tools. Linking results obtained by e.g. automatic first-order theorem provers or SAT solvers is often much harder than proving the requested goal by hand (things are much better in the new Isabelle/HOL version).

Appendix A

Dependency Graphs

To verify the disk driver, we had to integrate a wide range of theories written and developed by many people. Below, we present the dependcy graphs of the driver verification. Bubbles denote theories and an arrow $A \rightarrow B$ denotes that theory A imports theory B. To give an idea of the complexity of the integrated proof corpus, we sketch in Figure A.2 the transitive closure of the imported theories (the large circle marks the driver theories). Figure A.1 shows those theories, which were imported directly by the driver verification theories (large rectangle at the bottom). Most importantly these are:

- VAMP ISA semantics and theories for bit-level operations, written a.o. by S. Tverdyshev and M. Daum.
- VAMP assembly semantics, theories to verify assembly code and the simulation theory between VAMP assembly and VAMP ISA written by T. In der Rieden, D. Leinenbach and A. Tsyban.
- C0 small step semantics, compiler correctness and a theory to deal with inline assembly code, written by D. Leinenbach and A. Tsyban.
- Constants and definitions imported from the page-fault handler correctness theories (written by A. Starostin) to ensure a correct embedding of driver correctness.

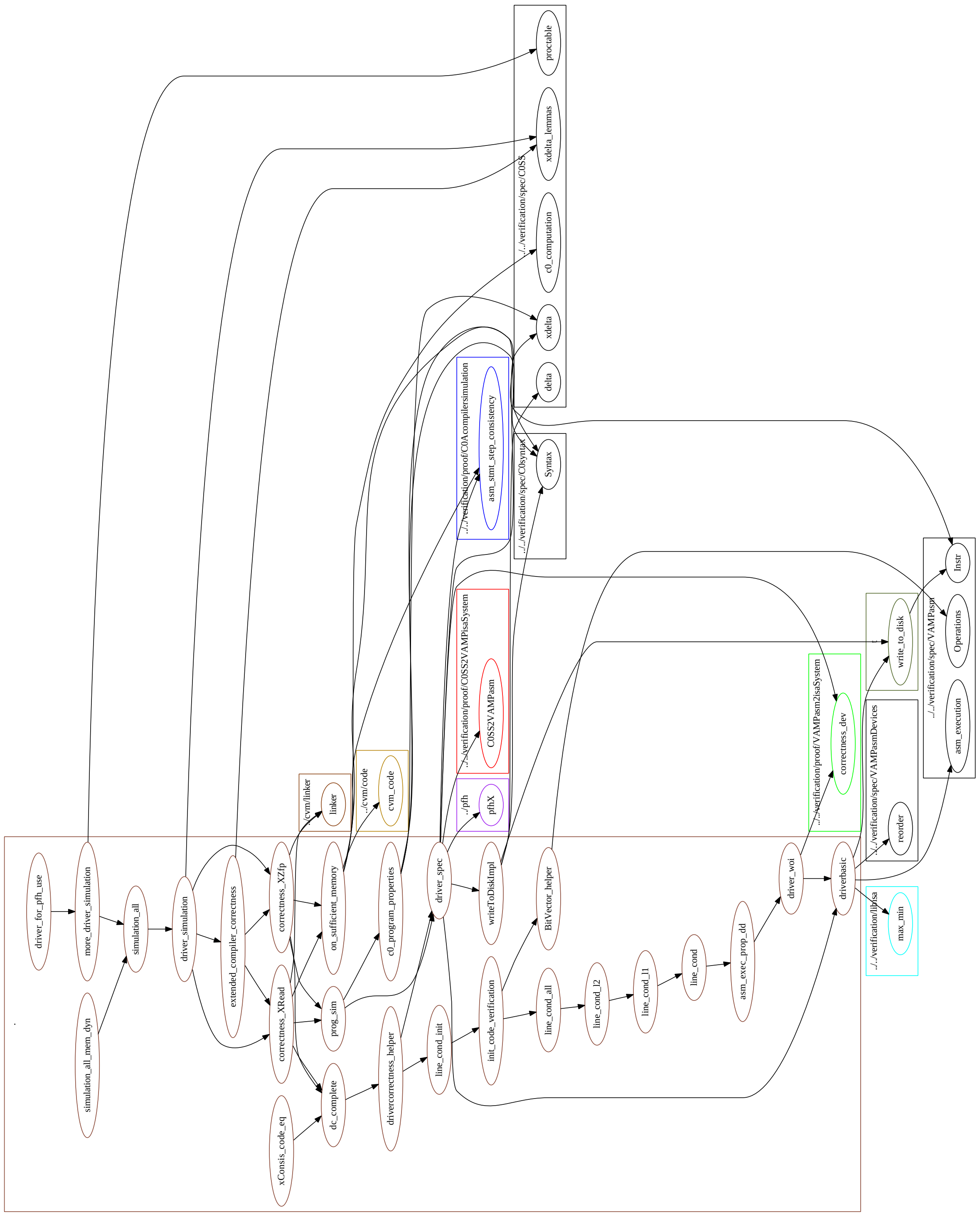


Figure A.1: Dependency graph for driver verification

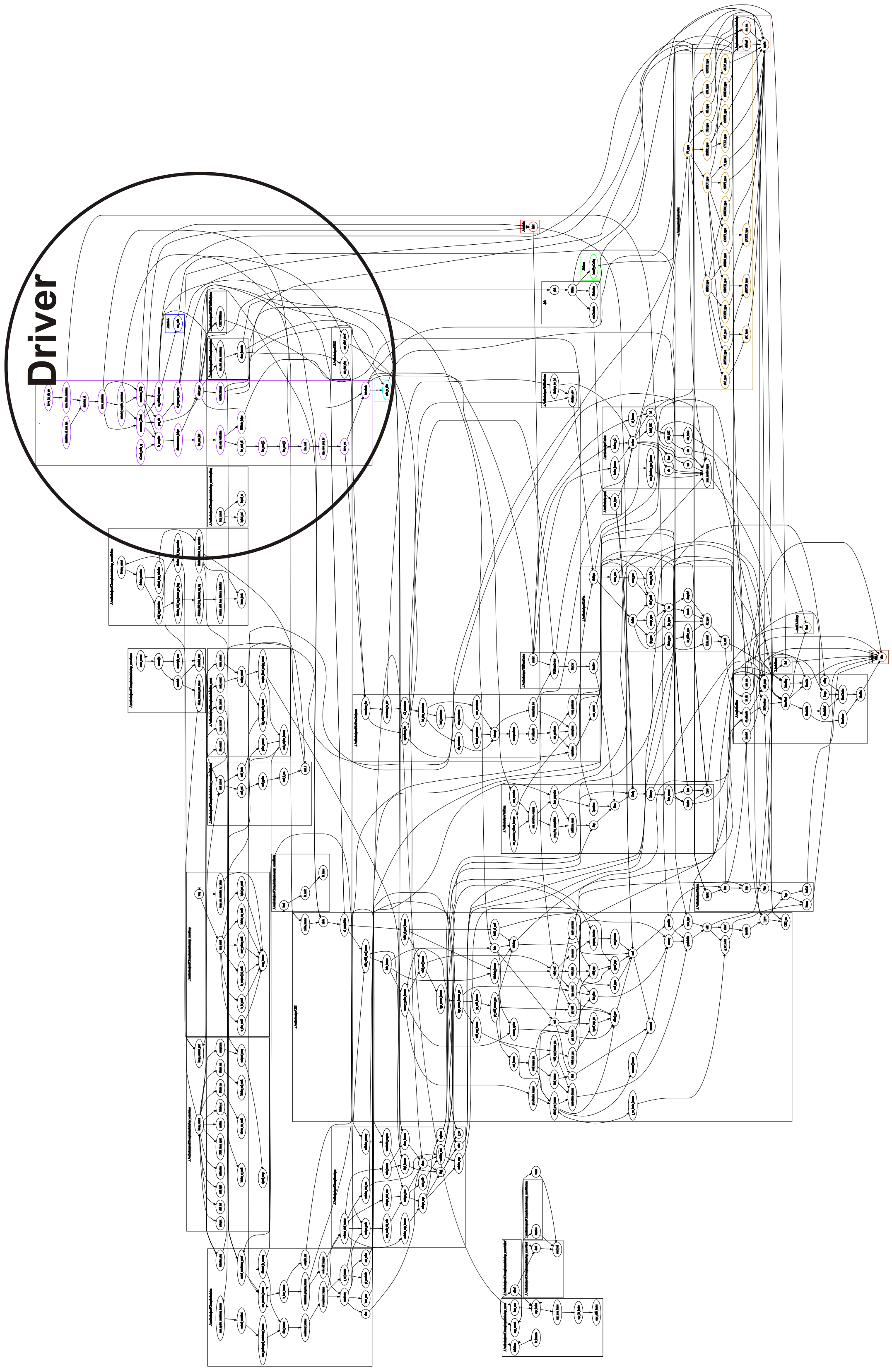


Figure A.2: Dependency graph for driver verification (transitive closure)

Appendix B

Mapping to Formal Names

This section gives a mapping from lemmas and theorems to the corresponding formal proofs in the Isabelle theories.

Name in document	Formal name
Lemma 1 (Well-defined filters)	prop_filter
Theorem 1 (VAMP assembly to VAMP ISA)	asm_simulates_isa_dev (A. Tsyban)
Lemma 3 (Non-Interference Observation)	commute_devices and commute_processor_device
Lemma 5 (No Device Access)	no_device_access_prop
Lemma 6 (Pure Sequences and Stable Devices)	reorder_empty_seq (instantiation for hard disk)
Theorem 2 (Reordering of Sequences)	reorder_seq
Theorem 3 (Non-interference of Devices)	devices_no_interference
Lemma 7 (Program Header Equivalence)	delta_first_statement_executes_invariant (D. Leinenbach)
Lemma 8 (Valid Property on Program Head)	valid_prop_hd_prop
Lemma 9 (Valid Property Invariant)	valid_prop_deltax_nth_invariant
Lemma 10 (SCalls Monotonicity)	Part of deltax_SCalls_invariant
Lemma 11 (SCalls Fixpoint)	SCalls_fix_point
Lemma 12 (Monotonicity of top-scalls on Transitions)	deltax_scalls_SCalls_invariant
Lemma 13 (Monotonicity of SCalls on Transitions)	deltax_SCalls_invariant
Lemma 14 (Valid Property Invariant 2)	deltax_valid_prop_SCalls_invariant
Lemma 15 (Increase of Stack Size)	Part of delta_stack_max_increase
Theorem 4 (Compiler Correctness)	compiler_correct (D. Leinenbach)
Theorem 5 (Extended Compiler Correctness)	compiler_correctness_device_extension

Lemma 18 (Monotonicity of Heap Consumption in C0)	delta_enough_heap_upper_bound
Theorem 6 (Upper bound for stack memory consumption)	sufficient_stack_size_frames_in_mem
Lemma 19 (Invariance of Stack-inv)	sufficient_stack_asize_memlist
Lemma 20 (Establishing the stack invariant)	Part of sufficient_stack_size _frames_in_mem
Lemma 21 (Relation of stack-inv and local-sz)	Part of sufficient_stack_asize_memlist
Lemma 22 (Control Consistency for Inline Assembly)	Part of parameter_passing_correct
Lemma 23 (Consistent Update)	consistent_C0_conf_asm_update (A. Tsyban)
Lemma 24 (Observation Valid on Sequence Set)	valid_ob_SeqV
Theorem 9 (C0 with XCalls compiler correctness)	simulation_XCalls_CVM (less general, instantiation to XCalls used in CVM)
Theorem 10 (Assembly driver correctness)	correctness_of_write_to_disk
Lemma 25 (Stability of hard disk)	device_idle
Lemma 26 (Addition)	addi_execution_woi (D. Leinenbach)
Lemma 27 (Shift as modulo/ And as division)	Part of construct_lba_from_reg_correct
Lemma 28 (Correctness of lba computation)	compose_lba_correct
Lemma 29 (Correctness of outer loop)	outer_loop
Lemma 30 (Correctness of first inner loop)	inv_loop1_complete
Lemma 31 (Smallest element)	hd_fair
Lemma 32 (During Polling)	l_loop2_inv
Lemma 33 (End of Polling)	l_loop2_rdy
Lemma 34 (End of Polling 2)	l_loop2_finish
Theorem 11 (Driver XCalls Compiler Correctness)	simulation_driver_zfp_pfh_use
Lemma 35 (Correctness of writePage implementation)	simulation_XCall_write

Bibliography

[ABK08] Eyad Alkassar, Peter Böhm, and Steffen Knapp. Correctness of a fault-tolerant real-time scheduler algorithm and its hardware implementation. In *Formal Methods and Models for Codesign (MEMOCODE'2008)*, pages 175–186. IEEE Computer Society Press, 2008.

[ABP09] Eyad Alkassar, Sebastian Bogan, and Wolfang Paul. Proving the correctness of client/server software. *Sādhanā Journal: Academy Proceedings in Engineering Sciences*, 34, 2009. To appear.

[AH08] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 225–239, Toronto, Canada, October 2008. Springer.

[AHK$^+$07] Eyad Alkassar, Mark Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. Formal device and programming model for a serial interface. In Bernhard Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, pages 4–20. CEUR-WS.org, 2007.

[AHL$^+$08] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224, Toronto, Canada, October 2008. Springer.

[AHL$^+$09] Eyad Alkassar, Mark A. Hillebrand, Dirk C. Leinenbach, Norbert W. Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load - leveraging a semantics stack for systems verification. *JAR: Special Issue on Operating Systems Verification*, 2009. To appear.

[Ame00] American National Standards Institute. *ANSI NCITS 340-2000: AT Attachment - 5 with Packet Interface*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 2000.

[ASS08] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In C. R. Ramakrishnan and Jakob Rehof, editors, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2008.

[BBBW07] Gerd Beuster, Thorsten Bormer, Pia Breuer, and Markus Wagner. Code-level verification of an email client. `http://www.verisoft.de/.rsrc/VerisoftRepository/vemail-trunk-r15868.tar.gz`, 2007.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[BHW06] Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification – Experiences from the Verisoft email client. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006)*, volume 192 of *CEUR Workshop Proceedings*, pages 112–125. CEUR-WS.org, August 2006.

[BJK$^+$03] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In Daniel Geist and Enrico Tronci, editors, *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2003.

[BJK$^+$06] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006.

[BKS03] Gérard Berry, Michael Kishinevsky, and Satnam Singh. System level design and verification using a synchronous language. In *ICCAD*, pages 433–440, 2003.

[BMS96] Manfred Broy, Stephan Merz, and Katharina Spies. The rpc-memory case study: A synopsis. In *Formal Systems Specification, The RPC-Memory Specification Case Study (the book grow out of a Dagstuhl Seminar, September 1994)*, pages 5–20, London, UK, 1996. Springer-Verlag.

[BN84] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[Bog08] Sebastian Bogan. *Formal Specification of a Simple Operating System.* PhD thesis, Saarland University, Computer Science Department, August 2008.

[Böh07] Peter Böhm. Formal verification of a clock synchronization method in a distributed automotive system. Master's thesis, Dept. of Computer Science, Saarland University, 2007.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01*, pages 103–122, 2001.

[BW07] Andrew Butterfield and Jim Woodcock. Formalising Flash memory: First steps. In *ICECCS*, pages 251–260. IEEE Computer Society, 2007.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *CONCUR '98*, pages 317–331, London, UK, 1998. Springer.

[CMST09] Ernie Cohen, Michal Moskał, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009.

[Coh00] Ernie Cohen. Separation and reduction. In *MPC'00*, pages 45–59. Springer, 2000.

[Dau08] Matthias Daum. Modelling user programs on top of a microkernel. In Elena Troubitsyna, editor, *Proceedings of Doctoral Symposium held in conjunction with Formal Methods 2008*, volume 48 of *General Publications.* Turku centre for computer science, 2008.

[DDB08] Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings, 5th International Verification Workshop (VERIFY), Sydney, Australia*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, August 2008.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In Dominique Borrione and Wolfgang Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 2005.

[FFW07] Leo Freitas, Zheng Fu, and Jim Woodcock. POSIX file store in Z/Eves: An experiment in the verified software repository. In *ICECCS*, pages 3–14. IEEE Computer Society, 2007.

[FLI] The flint project. `http://flint.cs.yale.edu/flint/`.

[Hal07] Thomas C Hales. Jordan's proof of the jordan curve theorem, 2007.

[HEK$^+$07a] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 41(4):3–11, 2007.

[HEK$^+$07b] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 41(4):3–11, 2007.

[HIP05] Mark Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD '05*, pages 309–316. IEEE Computer Society, 2005.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew P. Tolmach. A principled approach to operating system construction in Haskell. In *ICFP*, 2005.

[Hol06] Gerard J. Holzmann. New challenges in model checking. In *Symposium on 25 years of Model Checking, Seattle, USA*, number 4925 in Lecture Notes in Computer Science. Springer, August 2006.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HP08] Mark A. Hillebrand and Wolfgang Paul. On the architecture of system verification environments. In Karen Yorav, editor, *Hardware and Software, Verification and Testing, Third International*

Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23–25, 2007, volume 4899 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2008.

[HTS02] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: The VFiasco project. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, pages 165–169, New York, NY, USA, 2002. ACM Press.

[IdR09] Thomas In der Rieden. *Verifying CVM - The Kernel Parts.* PhD thesis, Saarland University, Computer Science Department, 2009. To appear.

[IT08] Tom In der Rieden and Alexandra Tsyban. CVM – A verified framework for microkernel programmers. In *3rd intl Workshop on Systems Software Verification (SSV 2008)*, volume 217C of *Electronic Notes in Theoretical Computer Science*, pages 151–168. Elsevier Science B.V., 2008.

[Kle09] Gerwin Klein. Operating system verification – An overview. *Sādhanā: Academy Proceedings in Engineering Sciences*, 34, 2009. To appear.

[Kna08] Steffen Knapp. *The Correctness of a Distributed Real-Time System.* PhD thesis, Universität des Saarlandes, 2008.

[KP06] Steffen Knapp and Wolfgang Paul. Realistic worst case execution time analysis in the context of pervasive system verification. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, volume 4444, pages 53–81, 2006.

[Lei08] Dirk Carsten Leinenbach. *Compiler Verification in the Context of Pervasive System Verification.* PhD thesis, Saarland University, Computer Science Department, July 2008.

[Lev93] Nancy G. Leveson. An investigation of the therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LNRS07] Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan. Verification of distributed applications. In Francesca Saglietti and Norbert Oster, editors, *Computer Safety,* *Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18–21, 2007*, volume 4680 of *Lecture Notes in Computer Science*, pages 315–328. Springer, 2007.

[Lyn96] Nancy A Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mic04] Microsoft Corporation. SDV: Static driver verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx, 2004.

[Moo03] J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2003.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NF03] Peter G. Neumann and Richard J. Feiertag. PSOS revisited. In *ACSAC '03*, pages 208–216, 2003.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NYS07] Zhaozong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *TPHOLs '07*, pages 189–206. Lecture Notes in Computer Science, 2007.

[RPS01] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-Chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[SC06] Scott D. Stoller and Ernie Cohen. Optimistic synchronization-based state-space reduction. *Form. Methods Syst. Des.*, 28(3):263–289, 2006.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich, April 2006.

[SCR$^+$03] S Shepler, B Callaghan, D Robinson, R Thurlow, C Beame, M Eisler, and D Noveck. RFC 3530: Network file system (nfs) version 4 protocol, 2003.

[Sha06] Andrey Shadrin. Design and implementation of the portmapper and RPC primitives in the context of the SOS. Master's thesis, Saarland University, Saarbrücken, 2006.

[Sri95] R Srinivasan. RFC 1831: RPC: Remote procedure call protocol specification version 2, 1995.

[Sta06] Artem Starostin. Formal verification of a c-library for strings. Master's thesis, Saarland University, 2006.

[Sta09] Artem Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, Computer Science Department, 2009. To appear.

[TA08] Sergey Tverdyshev and Eyad Alkassar. Efficient bit-level model reductions for automated hardware verification. In *15th International Symposium on Temporal Representation and Reasoning (TIME 2008)*, pages 164–172. IEEE Computer Society Press, 2008.

[The09] The Verisoft Consortium. The Verisoft Project. http://www.verisoft.de/, 2009.

[TK04] Harvey Tuch and Gerwin Klein. Verifying the L4 virtual memory subsystem. In Gerwin Klein, editor, *Proceedings of the NICTA Formal Methods Workshop on Operating Systems Verification*, pages 73–97. National ICT Australia, 2004.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL '07*, pages 97–108. ACM Press, 2007.

[TR85] Andrew S Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17(4):419–470, 1985.

[TS08] Sergey Tverdyshev and Andrey Shadrin. Formal verification of gate-level computer systems. In Kristin Yvonne Rozier, editor, *LFM 2008: Sixth NASA Langley Formal Methods Workshop*, NASA Scientific and Technical Information (STI), pages 56–58. NASA, 2008.

[Tsy09] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Computer Science Department, 2009. To appear.

[Tve09] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Computer Science Department, 2009. To appear.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[YS04] Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In *ICFP '04*, September 2004.