Formal Verification of a Framework for Microkernel Programmers

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Alexandra Tsyban

azul@wjpserver.cs.uni-sb.de

Saarbrücken, August 2009

Tag des Kolloquiums:	24. November 2009
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Reinhard Wilhelm
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Bernhard Beckert
akademischer Mitarbeiter:	Dr. Dirk Leinenbach

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im August 2009

This thesis work was founded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.

Abstract

This thesis presents the formal verification of a framework for microkernel programmers called CVM (communicating virtual machines) [41].

CVM is a computational model for concurrent user processes interacting with a generic microkernel and devices. It is implemented in $C0_A$, a restricted C-dialect with support of inline assembly, as a framework featuring virtual memory, demand paging, memory management, and low-level inter-process and devices communications. The framework can be linked on the source code level with an abstract kernel, an interface to users, in order to obtain a concrete kernel, a program that can be translated and run on a target machine. We use a formally verified microprocessor VAMP [20] as a platform to run the concrete kernel.

The main result of this work is a mechanically checked formal proof that concurrent executions of user processes interacting with a kernel are simulated by executions of the VAMP instruction set architecture model interleaved with devices. In order to obtain this result a number of attendant formal theories have been developed, most notably, a theory of inline assembly verification.

This work is a part of the Verisoft project [111], a large scale effort bringing together industrial and academic partners to push the state-of-the-art in formal verification for realistic computer systems comprising hard- and software.

Kurzzusammenfassung

Diese Arbeit präsentiert die formale Verifikation einer Umgebung für μ-Kernel Programmierer namens CVM (communicating virtual machines) [41].

CVM ist ein Berechnungsmodell für gleichlaufende Softwareprozesse, die mit einem generischen μ-Kernel und Geräten interagieren. Das Modell ist in einem beschränkten C-Dialekt mit Unterstützung von Inline-Assembly $C0_A$ implementiert und stellt die grundlegenden Funktionalitäten eines Betriebssystemkerns zur Verfügung: virtueller Speicher, Speicherverwaltung und rudimentäre Kommunikation zwischen Prozessen und Geräten. CVM wird auf der Quellcode-Ebene mit einem abstrakten Kernel verbunden, der eine Benutzerschnittstelle darstellt. Das Ergebnis ist ein konkreter Kernel, der kompiliert und auf der Hardware ausgeführt wird. Wir verwenden einen formal verifizierten μ-Prozessor VAMP [20] als Zielarchitektur.

Das wichtigste Ergebnis dieser Arbeit ist ein mechanisch geprüfter formaler Beweis. Dieser besagt, dass das Prozessormodell mit Geräten gleichlaufender Softwareprozesse, die mit einem Kernel interagieren, simuliert. Um dieses Ergebnis zu bekommen, wurde eine formale Theorie entwickelt, die insbesondere die Inline-Assembly Verifikation umfasst.

Diese Arbeit ist Teil des langfristig angelegten Forschungsprojektes Verisoft [111]. Das Projekt bringt die industriellen und akademischen Partner zusammen, um die Technologie der formalen Verifikation für realistische Computersysteme weiter zu entwickeln.

Contents

Chapter 1

Introduction

Contents

1.1 Motivation and Background

As long as requirements to computer designs are formulated in an ambiguous human language and as long as these designs are implemented by humans not insured against possible carelessness computer systems will contain errors. For the time being, the only way to guarantee absence of errors in a computer system is to exploit rigorous formal methods of mathematics for specifying system's intended behavior and ensuring that the actual system's implementation meets the desired behavior. The latter is known as *formal verification*. There are two main approaches to formal verification: model checking, a systematical exhaustive exploration of the problem's mathematical model, and theorem proving, a formal mathematical reasoning about a system. While the first method is fully automated, the second approach requires user's investigations for conducting a proof. The proof is then examined automatically by hopefully sound proof checkers. However, a technology of today allows us to apply model checking to a significantly smaller set of problems compared to theorem proving.

A program proven correct in a high-level programming language may not execute as expected on a particular computer. Such correctness proof ignores irregular patterns of control flow which take place due to multitasking and interrupts on the computer. High-level data types and operations used to implement the program and formulate its

correctness criteria differ from flip-flops and signals that occur in the hardware. The gap between what has been proven about the program in the high-level language semantics and what is actually executed on the underlying hardware may be a source of errors.

The solution to the problem is to verify the execution environment of the program: the operating system to ensure correct assignment of hardware resources to the program and non-interference with other programs, the compiler and assembler to guarantee correct translation from high-level data types and operations to the machine instruction level, the actual hardware implementation to make certain that it meets the instruction set architecture. The approach is called *systems* or *pervasive* verification and was introduced in 1989 by Bevier, Hunt, Moore, and Young in [17].

In order to ensure that interfaces of all components of the program's execution environment fit together a common formal framework has to be used. By choosing the implementation model of each layer to be the specification of the next lower layer it is possible to combine the components into a verified stack. With the program on top of the stack one achieves the highest degree of assurance in program correctness.

1.2 Microkernels

It is fair to put an operating system at the heart of a hardware-software stack. Managing computer physical resources, like CPU time and memory, and providing users with interfaces to these resource, operating system are the actual bridge between the hardware and applications. The core part of an operating system that is executed in processor's supervisor mode and has full access to hardware resources is called a *kernel*. It provides basic means for operating systems functionality: memory management, interrupt handling, inter-process communication, device drivers, etc. A *microkernel* is a kernel based on the principle of minimality of code and concepts.

The history of microkernels goes back to the 1970's. Conceptually the first microkernel system was the Nucleus [44] designed by Hansen. It featured primitives for process control and inter-process communication moving all operating system policies outside into a special user process. The idea evolved in the Hydra system [68] — though, the term *microkernel* itself has been introduced in the eighties to describe the Mach system [96]. Late eighties gave birth to lots of microkernels: most notable were the QNX [48] and Chorus [100] systems. In fact, all eighties designs fell short to meet the minimality requirements of microkernels. For instance, the Mach system featured over 150K lines of code implementing over 200 of system calls.

In the early nineties microkernels received severe criticism [23] for their poor performance caused by frequent user-kernel mode and address-space switches. Later on, Liedtke managed to re-analyze the performance of Unix on Mach compared to native monolithic Unix and showed that the efficiency of the system based on the Mach kernel was limited by cache misses [70] — the kernel was too big in size.

With second generation of microkernels Liedtke showed [70, 71, 72] that the performance issues could be resolved if the microkernel minimality principle is taken earnestly. By strongly simplifying microkernel concepts, taking very careful approach to design and implementation, and extensively optimizing underlying algorithms and data structures [69] he developed the first L4 kernel. Primarily designed with high performance in mind, L4 was written in assembly language. The L4Ka project [95] organized by Liedtke in 1999 showed that high-performance microkernels could be implemented in a high-level programming language. As a proof of concept the group developed L4Ka::Hazelnut, a C++ version of the kernel that ran on IA32- and ARM-based machines. The developed

kernel was an oder of magnitude smaller than first-generation microkernels: it featured only about 10K lines of code. The L4 microkernel motivated creation of a number of clones as well as ports to different hardware platforms. The framework for microkernel programmers considered in this thesis is also inspired by L4.

1.3 Related Work

The related work of the thesis is summarized in two categories: (i) operating-system microkernel verification, and (ii) pervasive systems verification. For an in-depth retrospective of the first topic we recommend the reader to consult Klein's article [63] which provides an outstanding overview of the subject.

1.3.1 Operating-System Microkernel Verification

First attempts to use theorem provers for formal specification and correctness proofs of operating systems date back to the mid seventies in PSOS and UCLA DSU projects.

Provably Secure Operating System (PSOS) was designed at SRI International [58] in 1973-1980 as a general-purpose operating system with provable security properties. Neumann and Feiertag provide a retrospective view of the system in [80] — earlier reports include [38] and [81]. Founded on capability-based security, the design of PSOS provided an early example of hierarchically layered abstraction. The hardware-software architecture was type-safty. SPECIfication and Assertion Language (SPECIAL) [99] was used to precisely specify each module at each layer as well as interlayer abstraction mappings. A number of application layers were also formally specified. As for verification, formally provable trustworthiness of the system and its applications was a far-reaching goal at that time. Only simple illustrative proofs were carried out to demonstrate how properties could be formally proven — *in the sense that the specification could be formally consistent with the requirements, the source code could be formally consistent with the specifications, and the compiler could be proven correct as well* — to cite the authors themselves.

UCLA Data Security Unix (DSU), a kernel-structured operating system, was developed at the University of California at Los Angeles (UCLA) [86] in the late seventies in order to demonstrate that program verification methods could be applied to prove an operating system secure. Walker, Kemmerer, and Popek report in [117] on the specification and verification experience of DSU. Data Security Unix was implemented in Pascal as a multiprogramming uni-processor operating system running on a DEC PDP-11/45 computer, with application interface similar to standard Unix. The kernel supported processes, capabilities, pages and non-modeled devices via kernel calls. The verification objective was a data security proof: direct access to data must be granted only if the recorded protection policy permits it. For the proof that the kernel is secure four levels of specification ranging from Pascal code to the top-level security property were conducted in XIVUS [42], a verification system based on the first-order predicate calculus. A proof that specifications on that different levels of abstraction are consistent with each other was undertaken but not completed for all portions of the kernel. The work assumed that the Pascal compiler and the hardware operate correctly.

From today's perspective, both projects achieved superficial verification results to a large extent due to underdeveloped verification environments as well as specification and proof techniques. A substantial progress in microkernel verification has been achieved in the late eighties with the KIT project.

Kernel for Isolated Tasks (KIT) was a small operating-system kernel written at the University of Texas at Austin [88] for a simple von Neumann computer architecture. KIT verification is described in the doctoral thesis of Bevier [15] — a short summary is [16]. The kernel implemented in a machine language services for process scheduling, error handling, single-word message passing, and character I/O to non-modeled devices. KIT lacked dynamic creation of processes as well as shared memory and demand paging. A top-level correctness property was process isolation: execution of one process must not interfere with the other in unintended ways. In order to establish the property a number of abstraction layers as well as simulation theorems connecting them were developed in the Boyer-Moore theorem prover [22]. The KIT project was the first example of an accomplished mechanically checked proof of the correct implementation of a complete, though suitable only for special-purpose systems, operating-system kernel.

The VFiasco project held at the Technical University of Dresden [87] aims at the formal verification of a small L4-compatible operating-system microkernel Fiasco. Hohmuth and Tews summarize the project details in [54]. The Fiasco kernel is implemented with less than 15K lines of source code. As an experiment the SPIN model-checker [55] was applied to certify a rudimentary version of Fiasco's IPC [35]. Unsatisfactory results were achieved due to the size of problem's state space. A a solution a theorem proving approach was undertaken: a subset of C++ was formalized in PVS [89] in order to reason about the implementation of Fiacso. The authors considered various jump statements like `break` and `goto` together with type casts that can turn integers into pointers as *two major features of C++* necessary to implement kernels. The denotational semantics for both cases [53, 109] was developed and applied to a case study [109]. The current status of Fiasco formal verification is vague.

The Coyotos team [26] has defined a new low-level programming language BitC with precise formal semantics in order to carry out verification of a general-purpose operating system Coyotos. Shapiro et al. elaborate on the Coyotos architecture and verification approach in [59]. The project is the successor to EROS [103], a high-performance capability-based operating system running on Pentium processors. Though the EROS team considered verification of kernel security properties [104] only in Coyotos project kernel correctness is the major issue. The Coyotos kernel is implemented in BitC, a language for system programmers developed in the scope of the project. Originated from Scheme [62], BitC is best viewed as ML [92] with machine-level representation types and C-style structures. The proposed verification objectives cover correctness proofs of address translation and memory safety over the kernel implementation. As yet, the status of formal verification is unclear.

The L4.verified project is carried out at the National ICT Australia (NICTA) [9]. In 2004-2006 the project focused on understanding and formalizing an L4 microkernel implementation L4Ka::Pistachio [73] for the ARM architecture [60]. An abstract model of address spaces, one of the three main abstractions of L4 together with threads and inter-process communication, has been built and refined against its C implementation. Correctness proofs we developed in Schirmer's verification environment for sequential imperative programs [101] embedded in Isabelle/HOL theorem prover [85]. Tuch and Klein report on the verification experience in [113]. The current research at NICTA is aimed at construction and verification of seL4 (secure embedded L4), an L4 kernel extended with a model of capabilities. Heiser et al. summarize their research on seL4 in [46]. From seL4 prototype designed in Haskel both formal model and high-performance C implementation are generated. The verification goal is to show in Isabelle/HOL that the produced implementation conforms with an abstract model. The project lacks a verified C compiler, however a detailed memory model for low-level pointer programs

in C which provides convenient separation logic abstraction [114] has been developed. For kernel verification it is supposed to show a refinement between three levels of abstraction: from C implementation — through executable specification — to the abstract kernel model. As of July 2009, the project has completed the formal correspondence proof between abstract and executable specifications and 95% of a simulation proof between the specification and C implementation is done.

The Singularity project has started in 2003 at the Microsoft Research [98] with the goal of examining shortcomings of existing systems and designing from the scratch a software platform with the primary goal of dependability. Hunt et al. summarize the project in [36] — as yet, the most recent project's state is described in [56]. The Singularity operating system is developed with three key architectural features in mind: software-isolated processes, contract-based channels, and manifest-based programs. 90% of the system is written in Sing# [37], a new type-safe, garbage-collecting programming language based on Spec# [11]. The remaining part is implemented in unsafe C++ and assembly languages. Spec# is an extension to Microsoft's C# [120] programming language that provides means (pre- and postcondition as well as object invariants) for specifying program behavior. Specification are either statically proved by the Boogie verifier [10] or checked by compiler-inserted run-time tests.

The FLINT group at the Yale University [116] focuses on studying separate problems in operating-system verification rather than proving a complete OS correct. Ni, Yu, and Shao report in [83] on their successful experience of applying XCAP [82] — a theoretical verification framework — to certify a realistic x86 assembly implementation of machine-context management procedures. XCAP follows Hoare logic [52] and is implemented in the Coq proof assistant [12]. In June 2008 Feng et al. presented in [39] a Hoare-logic-like framework for certifying low-level programs with hardware interrupts and preemptive threads. The work provides a solid foundation for reasoning about preemptive kernels and hypervisors.

The Robin project [94] has started in 2006 as a collaborating between Technical University of Dresden [87], Radboud University Nijmegen [84], and industrial partners. The project is supposed to develop a minimal trusted computing base — the Nova microhypervisor — for virtualizing multiple instances of conventional operating systems in a secure way. Tews reviews the project in [110]. Nova exploits the Intel Virtualization Technology [25] to virtualize legacy operating systems. The project exploits and further develops the Nizza [45] architecture. The hypervisor is implemented in a subset of C++. For verification a simplified x86 hardware model as well as semantics of a C++ subset are specified in PVS [89], where the refinement proofs are also planned to be conducted. So far, there were no reports on the status of formal proofs.

The goal of the Verisoft XT project [112] is to specify and verify industrial software, including the Microsoft hypervisor Hyper-V, a component of the Windows Server 2008 [74, 76], and the PikeOS [13], a microkernel-based real-time operating system made by SYSGO AG [108]. The core specification and verification tool of the project is the verifying C compiler (VCC) [75], a verifier for concurrent C being developed at Microsoft research, Redmond, USA, and the European Microsoft Innovation Center (EMIC), Aachen, Germany. VCC takes a program (annotated with function contracts, state assertions, and type invariants) and attempts to prove the correctness of these annotations. As of July 2009 the project has succeeded with adding of 13500 lines of annotations to the Microsoft hypervisor codebase. About 350 functions have been successfully verified [75].

1.3.2 Pervasive Systems Verification

An innovative approach for pervasive systems verification was undertaken with the CLI stack [17, 18] in the late eighties. The stack was built and mechanically verified by Computational Logic, Inc. [24]. The 1989 version of the stack comprised the following 6 layers: (i) a gate-level design for the FM8502 (a 32-bit version of the 16-bit FM8501 [118]) microprocessor, (ii) an operational semantics for the corresponding instruction set architecture (ISA), (iii) an operational semantics for the stack-based assembly language Piton [77], (iv) a simple assembly-implemented operating-system: the Kernel for Isolated Tasks (KIT) [15, 16], (v) operational semantics for two toy high-level languages: micro-Gypsy [121] (a derivative of Pascal) and a subset of Lisp [40], (vi) a user application: the game NIM [119]. The layers were connected by various functions, including an assembler, linker, and compilers. The stack was developed and verified using the Boyer-Moore theorem prover [22].

In 2003 Moore, the head of the CLI project, reviewed the stack and argued why it became impractical [78]. The reasons boil down to oversimplifications of the 1989 design. In particular, the processor had no pipeline and caches, the kernel lacked demand paging and the high-level languages were too simple to be of practical use. Devices were not consider through the whole stack. In the same paper Moore proposed a grand challenge for formal methods: a verification of a realistic stack.

In a response to the grand challenge the Verisoft project [111] has started in 2003. The project is a partnership between several German universities and industrial companies. The mission of the project is to develop the technology [90] which permits pervasive formal verification of realistic entire computer systems and to demonstrate this technology with several prototypes. One of the prototypes — *the academic system* — comprises the following layers, which are connected by respective simulation theorems: (i) a gate-level design of the VAMP [20], a RISC processor with out-of-order execution, caches [19], and memory-management units [50, 28, 27], (ii) an operational semantics for the DLX instruction set architecture [79], a derivative of the MIPS ISA [61], (iii) an operational semantics for the DLX assembly language, (iv) an operational semantics for the C0 programming language [67, 65, 66, 93], a slightly restricted dialect of C, (v) a framework for microkernel programmers, called Communication Virtual Machines (CVM) [41, 57], which implements low-level microkernel functionality including a page-fault handler [8, 105], context switch [107], communication [106] and memory-management primitives, (vi) an L4-inspired microkernel VAMOS [33, 29], which provides support for priority-based scheduling [31], user-mode device drivers, and interprocess communication (IPC) (vii) a user-level Simple Operating System (SOS) [21] featuring TCP/IP communication protocol and a file system, (viii) a number of useful user applications like remote procedure calls (RPC) [102], SMTP and signature servers, and an email client [14]. A decisive novelty of the project is integration of concurrent devices [3, 51, 64, 5], including a hard disk, through the whole stack. All work is conducted in Isabelle/HOL [85] theorem prover. In order to support management of formal theories a repository of verification environments [49] is built. As yet, all individual components of the stack, except for SOS, are verified, the simulation theorems between all layers are stated and formally proven between the four bottom layers.

1.4 Objective

This work is a part of the Verisoft project and has a goal to develop a feasible approach to pervasive formal verification of microkernel low-level functionality and to demon-

strate the feasibility by applying it to CVM, a framework for microkernel programmers. Communicating Virtual Machines is a computational model for concurrent user processes interacting with a generic microkernel and devices. CVM is implemented in C0 with inline assembly as a framework featuring virtual memory support, demand paging, memory management, and low-level inter-process, process-kernel, and devices communications. The framework can be linked on the source code level with an *abstract kernel*, an interface to users, in order to obtain a *concrete kernel*, a program that can run on a target machine, like the VAMOS kernel of Verisoft. Note that the abstract kernel is a parameter of CVM — thus, a range of concrete kernels can be built by instantiating the parameter with different interfaces.

The international industrial standard for computer security and correctness is called the Common Criteria (CC) [2] and is in effect since 1999. It provides a numerical grade of 7 Evaluation Assurance Levels (EAL1 – EAL7). As the level is higher, the higher confidence that the system's principle security features are reliably implemented is provided. Even the highest software certification level EAL7 currently requires machine checked formal proofs only for high-level designs of systems, e.g., their top-level specifications. As it follows from the related work (Section 1.3), the formal methods community believes that meeting EAL7 is not enough for the software to be trustworthy. Commonly, operating-systems verification projects go beyond these requirements and aim at machine checked proofs that actual implementations correspond to abstract specifications.

With this work we go even beyond simulation proofs between an implementation, commonly done in a high-level programming language, and specification. We aim at the pervasive formal verification of the framework for microkernel programmers. That means that we do not stop at providing formal evidences that the C0 with inline assembly implementation of the framework meets its abstract specification, rather we aim at the correctness theorem of the framework in terms of the underlying hardware model. As a result, our verification objective is a mechanically checked formal proof that concurrent executions of user processes interacting with a kernel implemented in C0 with inline assembly are simulated by executions of the VAMP instruction set architecture model interleaved with devices.

1.5 Tools

As pervasive formal verification involves lots of higher-order logic (HOL) abstractions and inductive simulation proofs between them we need an effective theorem proving environment for HOL. The theorem prover common in Verisoft is Isabelle/HOL [85].

Isabelle is an interactive theorem proving framework. It follows the LCF system [43] approach: it has a small logical core written in standard ML guaranteeing logical soundness. Isabelle is generic: it provides a meta-logic of the weak type theory to declare deductive systems. So far, the best developed logic is HOL, including a large mathematical library and various theories for definitional concepts like inductive sets, lists, primitive and well-founded recursions, etc. The main proof method of Isabelle is a higher-order version of resolution, based on higher-order unification. Though interactive, Isabelle also features efficient automatic decision procedures, such as a term rewriting engine, called the simplifier, and a tableaux prover, called the classical reasoner.

1.6 Foundations and Contributions

This work is based on a number of formal results achieved within Verisoft. We import the following formal theories: (i) the model of VAMP instruction set architecture formerly specified in PVS by Beyer [19] and translated to Isabelle/HOL by Tverdyshev [115], (ii) the theory of generic devices developed by Alkassar [5], and Knapp [64], (iii) the C0 small-step semantics and the C0 compiler correctness theorem developed by Leinenbach [66], and (iv) the page-fault handler correctness theorem developed by Starostin [105].

In the scope of this thesis the following formal theories are developed: (i) the VAMP assembly semantics together with the correctness theorem towards VAMP ISA, (ii) the ministack between C0 and VAMP ISA through the VAMP assembly, (iii) the VAMP inline assembly semantics for C0, which allows to switch computations from the C0 to assembly level, (iv) the formal specification of a linker for C0 programs in collaboration with In der Rieden, (v) correctness proof of the linker, (vi) the formal specification of CVM in collaboration with Gargano, Hillebrand, Leinenbach, Paul, Alkassar, Daum, In der Rieden, and Knapp, (vii) all correctness criteria of CVM except for the relation between the abstract and the concrete kernel, (viii) the correctness theorem of CVM together with its proof.

1.7 Document Organization

The remainder of this thesis is organized in nine chapters.

- In Chapter 2 we define the mathematical notation used in the thesis.
- Chapter 3 presents the computational models involved in the work, namely: VAMP ISA and assembly, theory of generic devices, combination of devices with processors, and C0 small-step semantics.
- In Chapter 4 we introduce a number of simulation theorems justifying correctness of execution C0 programs on ISA and assembly machines. We start with a simulation theorem between VAMP ISA and assembly, introduce C0 compiler correctness theorem, and show how to combine these theorems into a monolithic ministack from C0 down to ISA.
- In Chapter 5 we define the model of Communicating Virtual Machines.
- Chapter 6 elaborates on the implementation of CVM in C0 with inline assembly. We define a formal linking operator which allows to build concrete kernels from the CVM implementation and an abstract kernel. We prove correctness of linking as well.
- In Chapter 7 we discuss correctness criteria of Communicating Virtual Machines and state the CVM correctness theorem.
- In Chapter 8 we elaborate on the inline assembly semantics for C0, a formal approach to reason about the code of the CVM framework which is a mixture of C0 and VAMP assembly.
- Chapter 9 deals with verification of the CVM source code. We present formal proofs for separate CVM components like dispatcher, primitives, and process-context switch.

- In Chapter 10 we present the correctness proof of user-processes computations.
- We conclude in Chapter 11 and discuss the future work.

Certainly, all lemmas and theorems presented in this thesis are formally proven in Isabelle/HOL. However, we omit some paper-and-pencil proofs in this thesis in two cases: if the proof is not conducted in the frame of this thesis, but rather done by some colleague, or if the proof is simple and lacks interesting peculiarities. Almost for every single definition, lemma, or theorem developed in the framework of this thesis we provide a link to a formal theory in Isabelle in the Verisoft repository [97], i.e.,

Isabelle: ModuleName/TheoryName.DefinitionName.

Chapter 2

Notation

We denote the set of natural numbers including zero by $\mathbb{N}$, the set of integers by $\mathbb{Z}$, the set of boolean values $\{\mathsf{T}, \mathsf{F}\}$ by $\mathbb{B}$, and the set of identifiers, e.g., variable names, by $\mathbb{S}$. We write 2^t for the power set of t. We denote the maximum of two numbers m and n by $\mathtt{max}(m, n)$ and the minimum by $\mathtt{min}(m, n)$. Let $\lceil n \rceil_{\mathrm{k}}$ be n divided by k and rounded up:

$$\lceil n \rceil_{\mathrm{k}} \stackrel{\text{def}}{=} (n + k - 1)/k.$$

We denote the type of an abstract list with elements of type t as t^*. We write $[]$ for an empty list, and by $[x, y, \ldots]$ we construct a list of particular elements. To obtain a list of given length n where each element is equal to x we use the notation x^n. For concatenation of two lists xs and ys we write $xs \circ ys$. The function $|xs|$ returns the length (number of elements) of the list xs. We extend the *belongs to set* notation $\in$ for lists: for an element x and a list xs we write $x \in xs$ instead of $\exists i : \ xs[i] = x$. To filter a list xs we use the notation $[x_{xs} : \ P(x)]$. By such expression we obtain a new list which consists only of those elements from the given list xs for which the predicate P holds.

We represent bits with the type $Bit = \{1, 0\}$. A list of several bits is a bit vector, an instance of the type $Bv = Bit^*$. The leftmost bit is the most significant bit and the rightmost bit is the least significant bit.

For a bit vector w we denote by $\langle w \rangle$ the conversion to the natural number with binary representation w. For a natural number n we denote by $bin(n)$ the conversion to the binary representation of n. Similarly for the integers, $[w]$ converts a bit vector w to an integer with two's complement representation w. Conversion to the two's complement representation of integer i is denoted by $two(i)$.

Besides specific abstract data types we will define further in this thesis, we introduce a general *option* type. It is used to extend the existing type with some error value $\perp$. For a particular type t we write $t_\perp \ = \ t \ \cup \{\perp\}$ to define such extended type. All non-error values $x \in t$ will be written as $\lfloor x \rfloor \in t_\perp$.

We introduce the constant A to denote an undefined value.

Chapter 3

Basic Concepts

Contents

In this chapter we introduce the stack of computational models needed to define CVM and its correctness criteria. Our starting point is a hardware model: in Section 3.1 we specify the model of VAMP ISA at the bottom of the stack and then abstract it in Section 3.2 to the model of VAMP assembly. Section 3.3 defines the theory of generic devices while in Section 3.4 we show how to combine devices systems with processors. In the end, Section 3.5 describes the C0 programming language and its small-step semantics. We do not present every single detail of VAMP ISA, devices, and C0 models since they were not developed in the scope of this thesis. The reader should consult theses of Beyer [19], Dalinger [27], and Tverdyshev [115] for the VAMP ISA model, of Knapp [64] and Alkassar [5] for the devices model, and of Leinenbach [66] and Petrova [93] for C0. Note that in this chapter we do not introduce simulation relations connecting all presented computational layers. The appropriate relations as well as simulation theorems are introduced in Chapter 4.

3.1 VAMP Instruction Set Architecture

In this section we introduce the computation model of the VAMP instruction set architecture (ISA). VAMP ISA is based on DLX ISA, a RISC processor architecture designed by Hennessy and Patterson [47]. Essentially, DLX is a cleaned and simplified 32-bit MIPS architecture [61]. The hardware platform of Verisoft, the VAMP processor [20], implements the DLX instruction set. However, the VAMP ISA model is significantly more complex than classical DLX. For instance, it features mechanisms for operating-system

support: user and system mode, and address translation in user mode for virtual memory support. The formal specification of VAMP ISA has been originally conducted in PVS by Beyer [19] and Dalinger [27]. It was consecutively translated into Isabelle/HOL by Tverdyshev [115] who also applied several automated proof techniques in order to optimize size of proof scripts.

3.1.1 Preliminaries

The predicate $Bv_{\mathrm{n}}\surd(w)$ indicates that the bit vector w is of length n:

$$Bv_{\mathrm{n}}\surd(w) \stackrel{\mathrm{def}}{=} |w| = n.$$

For a bit vector w the function

$$\mathit{fill0}(w) \stackrel{\mathrm{def}}{=} 0^{32-|w|} \circ w$$

extends w to the length of 32 with leading zeros. The sign extension of w up to the length of 32 with (copies of) the most significant bit is defined as follows:

$$\mathit{sxt}(w) \stackrel{\mathrm{def}}{=} \begin{cases} 0^{32} & \text{if } w = [] \\ b^{32-|w'|} \circ w' & \text{if } w = b \circ w' \end{cases}.$$

The addition of bit vectors a and b modulo 2^n is defined as:

$$a +_n b \stackrel{\mathrm{def}}{=} \mathit{fill0}(\mathit{bin}(\langle a \rangle + \langle b \rangle \bmod 2^n)).$$

Register Files

Registers are modeled as bit vectors. 32 registers form a register file. We model register files as mappings from bit vectors to bit vectors:

$$\mathit{Regf}_{\mathrm{ISA}} \stackrel{\mathrm{def}}{=} Bv \mapsto Bv.$$

For a register file $r :: \mathit{Regf}_{\mathrm{ISA}}$ the predicate

$$\mathit{Regf}_{\mathrm{ISA}}\surd(r) \stackrel{\mathrm{def}}{=} \forall\, i :\ Bv_5\surd(i) \longrightarrow Bv_{32}\surd(r(i))$$

indicates whether r has the appropriate size.

Memories

Due to potential extensions to double-word floating-point instructions memories are modeled as mappings from bit vectors to pairs of bit vectors:

$$\mathit{Mem}_{\mathrm{ISA}} \stackrel{\mathrm{def}}{=} Bv \mapsto (Bv \times Bv).$$

For a double-word (a pair of bit vectors) ww we denote the higher bit vector as $ww.\mathit{high}$ and the lower one as $ww.\mathit{low}$. For a memory $m :: \mathit{Mem}_{\mathrm{ISA}}$ the predicate

$$\mathit{Mem}_{\mathrm{ISA}}\surd(m) \stackrel{\mathrm{def}}{=} \forall\, a :\ Bv_{29}\surd(a) \longrightarrow Bv_{32}\surd(m(a).\mathit{high}) \wedge Bv_{32}\surd(m(a).\mathit{low})$$

indicates whether m has the appropriate size. For a 32 bit-vector address a we retrieve a single word from the double-word addressable memory using the following notation:

$$m_{\mathit{word}}(a) \stackrel{\mathrm{def}}{=} \begin{cases} m(a[31:3]).\mathit{high} & \text{if } a[2] = 1 \\ m(a[31:3]).\mathit{low} & \text{otherwise} \end{cases}.$$

Table 3.1: Indices of ISA special purpose registers.

Bin. index	Dec. index	Alias	Name
00000	0	*sr*	Status register
00001	1	*esr*	Exceptional status register
00010	2	*eca*	Exceptional cause
00011	3	*epc*	Exceptional program counter
00100	4	*edpc*	Exceptional delayed program counter
00101	5	*edata*	Exceptional data
01001	9	*pto*	Page table origin
01010	10	*ptl*	Page table length
01011	11	*emode*	Exceptional mode
10000	16	*mode*	Mode

3.1.2 Configurations

Configurations of the VAMP ISA model are defined as the record type C_{ISA}. An instance c_{ISA} has the following components:

- the normal $c_{\mathrm{ISA}}.pc :: Bv$ and delayed $c_{\mathrm{ISA}}.dpc :: Bv$ program counters used to specify the delayed branch mechanism (cf. Chapter 4 of [79]),
- the general purpose register file $c_{\mathrm{ISA}}.gpr :: \mathit{Regf}_{\mathrm{ISA}}$ and the special purpose register file $c_{\mathrm{ISA}}.spr :: \mathit{Regf}_{\mathrm{ISA}}$, and
- the memory $c_{\mathrm{ISA}}.m :: \mathit{Mem}_{\mathrm{ISA}}$.

We will also call VAMP ISA configurations *VAMP ISA machines*.

According to the DLX computational model the general purpose register zero always contains the zero value. In order to maintain this property we define the read and write functions over the GPR file.

Definition 3.1 (ISA GPR read) Reading from an ISA general purpose register file $r :: \mathit{Regf}_{\mathrm{ISA}}$ at an index $i :: Bv$ is done with the function

$$\mathit{gpr\text{-}read}_{\mathrm{ISA}}(r,i) \stackrel{\mathrm{def}}{=} \begin{cases} 0^{32} & \text{if } i = 0^{32} \\ r(i) & \text{otherwise} \end{cases}.$$

Isabelle: `VAMPasm2isaSystem/equivalence.GPRs_read`

The special purpose register file contains registers needed to process interrupts and registers used for virtual memory support. In the specification of ISA considered in this thesis not all of the 32 special purpose registers are used. Some of the special registers are just reserved for further possible extensions, e.g., an integration of a floating point unit. Table 3.1 defines the set $\mathit{sprs}_{\mathrm{ISA}} :: 2^{Bv}$ of special purpose register binary indices which we use.

Definition 3.2 (ISA SPR read) Reading from an ISA special purpose register file

$r :: Regf_{\text{ISA}}$ at an index $i :: Bv$ is done with the function

$$spr\text{-}read_{\text{ISA}}(r, i) \stackrel{\text{def}}{=} \begin{cases} r(i) & \text{if } i \in sprs_{\text{ISA}} \\ 0^{32} & \text{otherwise} \end{cases}.$$

Isabelle: `VAMPasm2isaSystem/equivalence.SPRs_read`

In order to express the well-formedness of ISA configurations we introduce the following predicate.

Definition 3.3 (Valid ISA configuration) Let c_{ISA} be an instruction set architecture model configuration. The predicate

$$\begin{aligned} isa\surd(c_{\text{ISA}}) \stackrel{\text{def}}{=} & \; Bv_{32}\surd(c_{\text{ISA}}.pc) \\ \wedge & \; Bv_{32}\surd(c_{\text{ISA}}.dpc) \\ \wedge & \; Regf_{\text{ISA}}\surd(c_{\text{ISA}}.gpr) \\ \wedge & \; Regf_{\text{ISA}}\surd(c_{\text{ISA}}.gpr) \\ \wedge & \; Mem_{\text{ISA}}\surd(c_{\text{ISA}}.m) \end{aligned}$$

states the validity requirements on c_{ISA}.

Isabelle: `VAMPasm2isaSystem/config_correct.is_dlx_conft`

3.1.3 Instructions

VAMP supports a variety of instructions for (i) memory, data transfer and control operations, (ii) arithmetic, logical, test, set and shift operations, and (iii) special operations for systems calls and return from exception. Table 3.2 depicts supported VAMP instructions.

Next, we sketch the predicates defined for specification of the instruction encoding. For complete definitions cf. [79] [91]. The instruction executed in configuration c_{ISA}, denoted by $iw(c_{\text{ISA}})$, is the memory word addressed by the delayed program counter. The six higher-order bits of the instruction word define the operation code (*opcode*):

$$opc(c_{\text{ISA}}) \stackrel{\text{def}}{=} iw(c_{\text{ISA}})[31:26].$$

VAMP instructions are grouped in three types.The instruction type defines how the instruction part outside the opcode is interpreted (cf. Figure 3.1). The predicates *is-rtype*(c_{ISA}), *is-itype*(c_{ISA}), and *is-jtype*(c_{ISA}) denote the type of the instruction $iw(c_{\text{ISA}})$.

We define the functions for extraction of individual instruction fields, e.g., the immediate constant is retrieved as

$$imm(c_{\text{ISA}}) \stackrel{\text{def}}{=} \begin{cases} sxt(iw(c_{\text{ISA}})[15:0]) & \text{if } is\text{-}itype(c_{\text{ISA}}) \\ fill0(iw(c_{\text{ISA}})[10:6]) & \text{if } is\text{-}rtype(c_{\text{ISA}}) \\ sxt(iw(c_{\text{ISA}})[25:0]) & \text{if } is\text{-}jtype(c_{\text{ISA}}) \\ 0^{32} & \text{otherwise} \end{cases}.$$

Depending on the instruction type, instruction word fields for destination (*rd*) and source registers (rs_1 and rs_2) have different positions. The functions for retrieving these com-

Table 3.2: Supported VAMP assembly instructions.

Data transfer	Arithmetic	Logical	Shift
`lb` (rd, rs, imm)	`addio`(rd, rs, imm)	`andi`(rd, rs, imm)	`slli`(rd, rs, sa)
`lh` (rd, rs, imm)	`addi` (rd, rs, imm)	`ori` (rd, rs, imm)	`srli`(rd, rs, sa)
`lw` (rd, rs, imm)	`subio`(rd, rs, imm)	`xori`(rd, rs, imm)	`srai`(rd, rs, sa)
`lbu`(rd, rs, imm)	`subi` (rd, rs, imm)	`lhgi`(rd, imm)	`sll` (rd, rs_1, rs_2)
`lhu`(rd, rs, imm)	`addo` (rd, rs_1, rs_2)	`and` (rd, rs_1, rs_2)	`srl` (rd, rs_1, rs_2)
`sb` (rd, rs, imm)	`add` (rd, rs_1, rs_2)	`or` (rd, rs_1, rs_2)	`sra` (rd, rs_1, rs_2)
`sh` (rd, rs, imm)	`subo` (rd, rs_1, rs_2)	`xor` (rd, rs_1, rs_2)	
`sw` (rd, rs, imm)	`sub` (rd, rs_1, rs_2)	`lhg` (rd, rs)	
Test		Control	Special move
`clri`(rd)	`clr`(rd)	`beqz`(rs, imm)	`movs2i`(rd, sa)
`sgri`(rd, rs, imm)	`sgr`(rd, rs_1, rs_2)	`bnez`(rs, imm)	`movi2s`(sa, rs)
`seqi`(rd, rs, imm)	`seq`(rd, rs_1, rs_2)	`jr` (rs)	
`sgei`(rd, rs, imm)	`sge`(rd, rs_1, rs_2)	`jalr`(rs)	
`slsi`(rd, rs, imm)	`sls`(rd, rs_1, rs_2)	`j` (imm)	
`snei`(rd, rs, imm)	`sne`(rd, rs_1, rs_2)	`jal` (imm)	
`slei`(rd, rs, imm)	`sle`(rd, rs_1, rs_2)	`trap`(imm)	
`seti`(rd)	`set`(rd)	`rfe`	

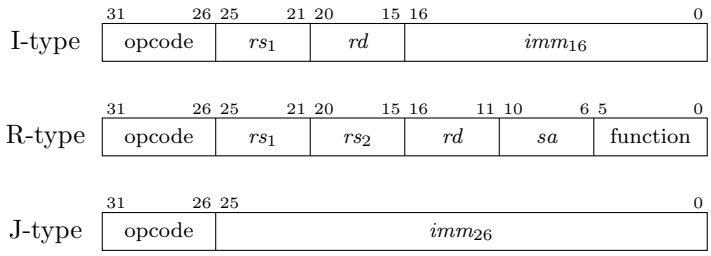

Figure 3.1: Instruction formats of the DLX computational model.

ponents define also case distinctions, for instance the destination operand is

$$rd(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \begin{cases} iw(c_{\mathrm{ISA}})[20:16] & \text{if } \textit{is-itype}(c_{\mathrm{ISA}}) \\ iw(c_{\mathrm{ISA}})[15:11] & \text{if } \textit{is-rtype}(c_{\mathrm{ISA}}) \\ 0^5 & \text{otherwise} \end{cases}.$$

Instruction decoding is formalized by predicates on $iw(c_{\mathrm{ISA}})$. For instance, we state that the current instruction is *trap* by means of the predicate

$$\textit{is-iw-trap}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \textit{opc}(c_{\mathrm{ISA}}) = 111110.$$

In the same fashion predicates for all other instructions are defined. Moreover, we define group predicates, e.g., *is-iw-alu* which holds for all arithmetic and logical instructions, or *is-iw-mem* which holds if the current instruction is some kind of load or store operation. In the same manner only write access is denoted by *is-iw-write*. The predicates *is-iw-byte* and *is-iw-word* group memory instructions according to the access width.

3.1.4 Semantics

The semantics of execution without interrupt is given by the transition function $\delta^{\text{woi}}_{\text{ISA}} :: C_{\text{ISA}} \mapsto C_{\text{ISA}}$ which yields for a configuration c_{ISA} the next state $c'_{\text{ISA}} = \delta^{\text{woi}}_{\text{ISA}}(c_{\text{ISA}})$. The definition of $\delta^{\text{woi}}_{\text{ISA}}$ splits cases depending on the instruction to be executed. We will specify the case for executing the *load word* instruction. For the remaining cases cf. [91].

The effective address $ea(c_{\text{ISA}})$ of load/store instructions is computed as the sum of the content of the general purpose register with index rs_1 and the immediate field $imm(c_{\text{ISA}})$. The addition is done modulo 2^{32} with two's complement arithmetic:

$$ea(c_{\text{ISA}}) \stackrel{\text{def}}{=} \mathit{gpr\text{-}read}_{\text{ISA}}(c_{\text{ISA}}.\mathit{gpr}, rs_1(c_{\text{ISA}})) +_{32} \mathit{imm}(c_{\text{ISA}}).$$

The decisive effect of the *load word* instruction is that the general purpose register addressed by $rd(c_{\text{ISA}})$ is updated with the memory word read at the effective address $ea(c_{\text{ISA}})$[1]:

$$c'_{\text{ISA}}.\mathit{gpr}(rd(c_{\text{ISA}})) = c_{\text{ISA}}.m_{\mathit{word}}(ea(c_{\text{ISA}})).$$

The semantics distinguishes two execution modes: *system*, which is defined as

$$\mathit{is\text{-}sys\text{-}mode}_{\text{ISA}}(c_{\text{ISA}}) \stackrel{\text{def}}{=} c_{\text{ISA}}.\mathit{spr}(\mathit{mode}) = 0^{32},$$

and *user*, defined as

$$\mathit{is\text{-}user\text{-}mode}_{\text{ISA}}(c_{\text{ISA}}) \stackrel{\text{def}}{=} c_{\text{ISA}}.\mathit{spr}(\mathit{mode}) = 0^{31} \circ 1.$$

An execution mode defines visibility rules for special purpose registers: in user mode it is forbidden to read and write them. Therefore the semantics of the `movi2s` and `movs2i` instructions is defined only in system mode. Further, the *return from exception* instruction is not allowed in user mode. Moreover, a memory access is subject to address translation in user mode. Hence, the semantics of load/store instructions uses a translated effective address and all instructions are fetched at translated delayed program counter.

3.1.5 Address Translation

In user mode all addresses for instruction fetch as well as for load/store operations are translated with the help of two special purpose registers *pto* and *ptl*.

The virtual address space is divided into pages of size `PAGE_SIZE` $\stackrel{\text{def}}{=} 2^{12}$ bytes. Each virtual address *va* is split into a virtual page index $va[31:12]$ and a byte index $va[11:0]$ which is an offset within the page. The main data structure for address translation is the *page table* which resides in the processor memory. The page table origin register and the virtual page index specify one page table entry. Formally, the page table entry in configuration c_{ISA} for a virtual address *va* is:

$$\mathit{pte}_{\text{ISA}}(c_{\text{ISA}}, va) \stackrel{\text{def}}{=} c_{\text{ISA}}.m_{\mathit{word}}(c_{\text{ISA}}.\mathit{spr}(\mathit{pto})[19:0] \circ 0^{12} +_{32} va[31:12] \circ 0^2)$$

Each page table entry *pte* contains the physical page index $pte[31:12]$. It also contains the following information: (i) the valid bit $pte[11]$ which denotes whether the page resides in the physical memory, (ii) the protection bit $pte[10]$ which denotes whether the page is

[1] The real semantics is more involved but for the load word instruction it boils down to a simple formula.

allowed to be written, and (iii) the execution bit $pte[9]$ which denotes whether the page contains executable code.

A physical page index combined with a byte index yields the complete physical address. Formally, the translated address is:

$$pa_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va) \stackrel{\mathrm{def}}{=} pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[31:12] \circ va[11:0]$$

The page table length register is used to specify the amount of allocated virtual memory. In case the virtual address does not belong to the user memory address translation results in a *page table length exception*. Formally, in configuration c_{ISA} the page table length exception for a virtual address va is:

$$ptl\text{-}excp_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va) \stackrel{\mathrm{def}}{=} \langle va[31:12] \rangle > \langle c_{\mathrm{ISA}}.spr(ptl)[19:0] \rangle$$

There are some situations when the translated address should not be used, either because invalid data was used for the translation, or processor must forbid the attempted operation at this address. This happens if the memory which stores an instruction is not tagged as executable, or protected memory is accessed for writing, or even the page containing this address is not present in the physical memory. The next predicate tests whether a problem occurs during address translation of a virtual address va in the configuration c_{ISA}. Note that the flag *is-mw* indicates memory write access and the flag *is-fetch* indicates instruction fetch:

$$\begin{aligned} transl\text{-}excp_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va, is\text{-}mw, is\text{-}fetch) \stackrel{\mathrm{def}}{=}\ & ptl\text{-}excp_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va) \\ \vee\ & is\text{-}fetch \wedge pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[9] = 0 \\ \vee\ & is\text{-}mw \wedge pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[10] = 1 \\ \vee\ & pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[11] = 0. \end{aligned}$$

3.1.6 Interrupts

Computations of the VAMP ISA semantics could be broken by interrupt signals which might be internal or external. An example of an internal interrupt is an illegal instruction. The external interrupts are reset and those that are generated by external devices. The interrupts are numbered with indices from 0 to 31. The interrupts are classified according to the following criteria: (i) maskable or not maskabale, (ii) internal or external, and (iii) of repeat, continue, or abort type. Maskable interrupts can be ignored under software control. If an interrupts signal arrives during execution of some instruction i and it is of repeat type then the instruction i is repeated when the program execution is resumed. If the interrupt is a continue interrupt then the instruction that follows i in the program is executed after the interrupt handling. In the remaining case the program execution is aborted.

Table 3.3 depicts interrupts supported by the VAMP ISA model. The *illegal instruction* interrupt $is\text{-}ill(c_{\mathrm{ISA}})$ occurs if the bit pattern of the instruction word $iw(c_{\mathrm{ISA}})$ does not match any defined VAMP instruction, denoted by $is\text{-}illegal(c_{\mathrm{ISA}})$, or if in user mode one of the three forbidden instructions are attempted to be executed:

$$\begin{aligned} is\text{-}ill(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=}\ & is\text{-}illegal(c_{\mathrm{ISA}}) \\ \vee\ & is\text{-}user\text{-}mode_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \wedge \\ & \quad (\ is\text{-}iw\text{-}rfe(c_{\mathrm{ISA}}) \\ & \quad \vee\ is\text{-}iw\text{-}movi2s(c_{\mathrm{ISA}}) \\ & \quad \vee\ is\text{-}iw\text{-}movs2i(c_{\mathrm{ISA}})) \end{aligned}$$

Table 3.3: Interrupts of the VAMP ISA model.

Index	Name	Meaning	Maskable	External	Type
0	*reset*	Reset	No	Yes	Abort
1	*ill*	Illegal instruction	No	No	Abort
2	*mal*	Misaligned access	No	No	Abort
3	*pff*	Page fault on fetch	No	No	Repeat
4	*pfls*	Page fault on load/store	No	No	Repeat
5	*trap*	Trap / System call	No	No	Continue
6	*ovf*	Overflow	Yes	No	Continue
12..31	*eev[j]*	Device interrupts	Yes	Yes	Continue

The *misaligned access* exception $\mathit{is\text{-}mal}(c_{\mathrm{ISA}})$ is raised if the memory-access width does not match (i) the low-order bits of the effective address $\mathit{ea}(c_{\mathrm{ISA}})$ ($\mathit{is\text{-}dmal}(c_{\mathrm{ISA}})$), or (ii) the delayed program counter in case of instruction fetch ($\mathit{is\text{-}imal}(c_{\mathrm{ISA}})$):

$$\mathit{is\text{-}mal}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \mathit{is\text{-}imal}(c_{\mathrm{ISA}}) \vee \mathit{is\text{-}dmal}(c_{\mathrm{ISA}}),$$

$$\mathit{is\text{-}imal}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} c_{\mathrm{ISA}}.\mathit{dpc}[1] = 1 \vee c_{\mathrm{ISA}}.\mathit{dpc}[0] = 1,$$

$$\begin{aligned}\mathit{is\text{-}dmal}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \mathit{is\text{-}iw\text{-}mem}(c_{\mathrm{ISA}}) \wedge (\neg \mathit{is\text{-}iw\text{-}byte}(c_{\mathrm{ISA}}) \wedge \mathit{ea}(c_{\mathrm{ISA}})[0] = 1 \\ \vee \mathit{is\text{-}iw\text{-}word}(c_{\mathrm{ISA}}) \wedge \mathit{ea}(c_{\mathrm{ISA}})[1] = 1).\end{aligned}$$

The *page fault on fetch* (equivalently, instruction page fault) interrupt is raised in the user mode whenever the translation of the fetch address cannot be done:

$$\begin{aligned}\mathit{is\text{-}pff}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} & \neg \mathit{is\text{-}imal}(c_{\mathrm{ISA}}) \\ \wedge & \ \mathit{is\text{-}user\text{-}mode}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\ \wedge & \ \mathit{transl\text{-}excp}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, c_{\mathrm{ISA}}.\mathit{dpc}, \mathsf{F}, \mathsf{T}).\end{aligned}$$

The *page fault on load/store* (equivalently, data page fault) is similar to the page fault on fetch, but here the effective address of the load/store instruction is examined:

$$\begin{aligned}\mathit{is\text{-}pfls}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} & \neg \mathit{is\text{-}dmal}(c_{\mathrm{ISA}}) \\ \wedge & \ \mathit{is\text{-}user\text{-}mode}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\ \wedge & \ \mathit{is\text{-}iw\text{-}mem}(c_{\mathrm{ISA}}) \\ \wedge & \ \mathit{transl\text{-}excp}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, \mathit{ea}(c_{\mathrm{ISA}}), \mathit{is\text{-}iw\text{-}write}(c_{\mathrm{ISA}}), \mathsf{F}).\end{aligned}$$

The *trap* interrupt, denoted by $\mathit{is\text{-}trap}(c_{\mathrm{ISA}})$, occurs if the special instruction `trap` is executed: $\mathit{is\text{-}iw\text{-}trap}(c_{\mathrm{ISA}})$. It provides means for the system call mechanism.
The *overflow* interrupt, denoted by the predicate $\mathit{is\text{-}ovf}(c_{\mathrm{ISA}})$, is raised if (i) neither instruction misalignment nor page fault on fetch occur, (ii) an arithmetic instruction takes place, and (iii) the overflow bit of the ALU output is on:

$$\begin{aligned}\mathit{is\text{-}ovf}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} & \neg \mathit{is\text{-}imal}(c_{\mathrm{ISA}}) \\ \wedge & \ \neg \mathit{is\text{-}pff}(c_{\mathrm{ISA}}) \\ \wedge & \ \mathit{is\text{-}iw\text{-}alu}(c_{\mathrm{ISA}}) \\ \wedge & \ \mathit{ALU}(c_{\mathrm{ISA}}).\mathit{ovf} = 1.\end{aligned}$$

Here ALU is the function specifying the arithmetic-logical unit, and its component $ovf :: Bit$ is set in case of an overflow during the instructions `addio`, `subio`, `addo`, and `subo`.

It is defined only by the configuration c_{ISA} which of the internal interrupts occur in the current configuration. Conversely, external interrupts are modeled as an external input $eev :: Bv$ of length 19. It is a parameter to the next-state function of VAMP ISA:

$$\begin{aligned} \delta_{\mathrm{ISA}} &:: C_{\mathrm{ISA}} \times Bv \mapsto C_{\mathrm{ISA}}, \\ c'_{\mathrm{ISA}} &= \delta_{\mathrm{ISA}}(c_{\mathrm{ISA}}, eev). \end{aligned}$$

Interrupt signals raised in the configuration c_{ISA} are collected in the *cause* register $ca(c_{\mathrm{ISA}}, eev)$.

$$\begin{aligned} ca(c_{\mathrm{ISA}}, eev) \stackrel{\mathrm{def}}{=} eev \circ 0^6 \circ [&bool2bit(is\text{-}ovf(c_{\mathrm{ISA}})), bool2bit(is\text{-}trap(c_{\mathrm{ISA}})), \\ &bool2bit(is\text{-}pfls(c_{\mathrm{ISA}})), bool2bit(is\text{-}pff(c_{\mathrm{ISA}})), \\ &bool2bit(is\text{-}mal(c_{\mathrm{ISA}})), bool2bit(is\text{-}ill(c_{\mathrm{ISA}})), 0]. \end{aligned}$$

Here, $bool2bit :: \mathbb{B} \mapsto Bit$ defines a trivial conversion from booleans to bits:

$$bool2bit(b) \stackrel{\mathrm{def}}{=} \begin{cases} 1 & \text{if } b = \mathsf{T} \\ 0 & \text{if } b = \mathsf{F} \end{cases}.$$

The *masked cause* vector is computed as a bitwise conjunction of $ca(c_{\mathrm{ISA}}, eev)$ with the mask stored in the status register $c_{\mathrm{ISA}}.spr(sr)$. If interrupt i is maskable and $c_{\mathrm{ISA}}.spr(sr)[i] = 0$ then bit i is masked out:

$$mca(c_{\mathrm{ISA}}, eev)[i] \;=\; \begin{cases} 0 & \text{if } i \text{ is maskable } \wedge\; c_{\mathrm{ISA}}.spr(sr)[i] = 0 \\ ca(c_{\mathrm{ISA}}, eev)[i] & \text{otherwise} \end{cases}.$$

If at least one bit of $mca(c_{\mathrm{ISA}}, eev)$ is on the *jump to interrupt service routine* (JISR) signal is activated:

$$jisr(c_{\mathrm{ISA}}, eev) \stackrel{\mathrm{def}}{=} \exists\, i :\; mca(c_{\mathrm{ISA}}, eev)[i] = 1.$$

In case JISR is not activated we continue with uninterrupted transition:

$$c'_{\mathrm{ISA}} \;=\; \delta^{\mathrm{woi}}_{\mathrm{ISA}}(c_{\mathrm{ISA}}).$$

Otherwise, if the lowest raised interrupt is of continue type then the instruction is executed, which leads to state $\hat{c}_{\mathrm{ISA}}$:

$$\hat{c}_{\mathrm{ISA}} \;=\; \begin{cases} \delta^{\mathrm{woi}}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) & \text{if } \textit{continue} \text{ interrupt} \\ c_{\mathrm{ISA}} & \text{otherwise} \end{cases}.$$

Afterwards, the program counters are set to the start address of the interrupt service routine. We assume it starts at address 0:

$$\begin{aligned} c'_{\mathrm{ISA}}.dpc &= 0^{32}, \\ c'_{\mathrm{ISA}}.pc &= 0^{30}10. \end{aligned}$$

The exceptional versions of program counters and the status and mode registers are assigned their normal versions:

$$\begin{aligned} c'_{\mathrm{ISA}}.spr(edpc) &= \hat{c}_{\mathrm{ISA}}.dpc, \\ c'_{\mathrm{ISA}}.spr(epc) &= \hat{c}_{\mathrm{ISA}}.pc, \\ c'_{\mathrm{ISA}}.spr(esr) &= \hat{c}_{\mathrm{ISA}}.spr(sr), \\ c'_{\mathrm{ISA}}.spr(emode) &= c_{\mathrm{ISA}}.spr(mode). \end{aligned}$$

Register *eca* is set to the masked interrupt cause:

$$c'_{\mathrm{ISA}}.spr(eca) = mca(c_{\mathrm{ISA}}, eev).$$

Register *edata* stores the data needed for interrupt handling:

$$c'_{\mathrm{ISA}}.spr(edata) = edata(c_{\mathrm{ISA}}) = \begin{cases} c_{\mathrm{ISA}}.dpc & \text{if } \textit{is-imal}(c_{\mathrm{ISA}}) \vee \textit{is-pff}(c_{\mathrm{ISA}}) \\ imm(c_{\mathrm{ISA}}) & \text{if } \textit{is-iw-trap}(c_{\mathrm{ISA}}) \\ ea(c_{\mathrm{ISA}}) & \text{if } \textit{is-iw-mem}(c_{\mathrm{ISA}}) \\ \mathsf{A} & \text{otherwise} \end{cases}.$$

Finally, the mode and the status registers are set to zero:

$$\begin{aligned} c'_{\mathrm{ISA}}.spr(mode) &= 0^{32}, \\ c'_{\mathrm{ISA}}.spr(sr) &= 0^{32}, \end{aligned}$$

which characterizes the system execution:

$$\textit{is-sys-exec}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} c_{\mathrm{ISA}}.spr(mode) = 0^{32} \wedge c_{\mathrm{ISA}}.spr(sr) = 0^{32}.$$

Execution of an interrupt service routine ends with the *return from exception* instruction `rfe`. According to its semantics the normal versions of registers *pc*, *dpc*, *sr*, and *mode* are assigned their exceptional versions:

$$\begin{aligned} c'_{\mathrm{ISA}}.dpc &= c_{\mathrm{ISA}}.spr(edpc), \\ c'_{\mathrm{ISA}}.pc &= c_{\mathrm{ISA}}.spr(epc), \\ c'_{\mathrm{ISA}}.spr(sr) &= c_{\mathrm{ISA}}.spr(esr), \\ c'_{\mathrm{ISA}}.spr(mode) &= c_{\mathrm{ISA}}.spr(emode). \end{aligned}$$

3.2 VAMP Assembly

Experience of Verisoft shows that reasoning about VAMP programs on the ISA level is superfluously hard for a number of reasons. As a response to this problem we introduce a convenient abstraction which we call the VAMP assembly. We start this section by arguing why it is uncomfortable to work with VAMP programs on the ISA level. Next, we introduce high-level types for representing registers and memories. Using these types we define configurations and semantics of the VAMP assembly model. Finally, we introduce a notion of VAMP assembly programs. Note, that correctness of the VAMP assembly model is justified in Section 4.1 by the simulation theorem towards VAMP ISA.

3.2.1 Motivation

There are four main peculiarities of the VAMP ISA model that make arguing about programs unnecessarily hard:

- bit-vector encoding of instructions,
- bit-vector representation of operands and program counters,
- unwanted interrupts, and
- low-level specification of functional units close to their implementations.

As instructions are represented by bit vectors in the VAMP ISA model, a quite complex instruction-decoding scheme is involved. This results in unwanted tedious proofs for extracting and decoding instruction mnemonics, source and destination registers, as well as an immediate constant. In the VAMP assembly model we represent instructions with an abstract data type, such that each instruction is modeled by means of its own inductive constructor. Source registers, destination registers, and immediate constants are passed as numerical parameters to these constructors.

Since programs are usually intended to perform computations with numbers it is quite inconvenient to deal with operands represented as bit vectors, as it is done in the VAMP ISA model. A convenient user-friendly specification of a computation claims its result in a numerical form. Hence, reasoning about such computation involves endless conversions — therefore, lots of undesirable proof steps — from bit vectors to natural and integer numbers and vice versa. In the VAMP assembly model we represent operands, registers and program counters with natural and integer numbers.

During verification of most of the programs it has to be shown that their executions do not produce interrupts. At the VAMP ISA level we have to prove many conditions in order to show that the program does not cause unwanted interrupts. The VAMP assembly model is designed without interrupts and thus we get for free the absence of interrupts in programs we verify.

Last but not least, the formal specification of the VAMP instruction set architecture was conducted in Isabelle/HOL with an idea in mind to ease verification of the VAMP processor, i.e., to simplify the proof that VAMP implements the VAMP ISA. As a consequence specifications of functional units like shifters on the ISA level almost coincide with their low-level, nearly imperative-style implementations. It turns out that the simplicity of processor verification and the feasibly of reasoning about instruction effects are quite orthogonal issues. Effective program verification requires clear high-level declarative instruction semantics — therefore, definitions of functional units. We resolve this question in the VAMP assembly model.

3.2.2 Data Representation

We represent register contents on the assembly level with 32-bit natural and integer numbers. The following predicates introduce notions of VAMP assembly natural and integer number, i.e., those that fit into 32 bits.

Definition 3.4 (VAMP assembly natural number) Let x be a natural number. The predicate

$$\mathbb{N}_{32}\surd(x) \stackrel{\text{def}}{=} x < 2^{32}$$

indicates whether x is representable with 32 bits.

Isabelle: `VAMPasm/Types.asm_nat`

Definition 3.5 (VAMP assembly integer number) Let x be an integer number. The predicate

$$\mathbb{Z}_{32}\surd(x) \stackrel{\text{def}}{=} -2^{31} \leq x < 2^{31}$$

indicates whether x is representable with 32 bits.

Isabelle: `VAMPasm/Types.asm_int`

In a similar to the VAMP ISA model manner we define addition modulo n. For natural numbers a and b we overload the notation $+_n$ as follows:

$$a +_n b \stackrel{\text{def}}{=} a + b \bmod 2^n.$$

In order to stay consistent with the binary arithmetic of the underlying ISA model we define conversion functions between integers and naturals. These functions simulate numerically the conversion from integers to bit vectors and then to naturals and vice versa.

Definition 3.6 (Conversion from integers to naturals) A 32-bit integer number i is converted into a natural number by means of the function

$$i2n(i) \stackrel{\text{def}}{=} \begin{cases} i + 2^{32} & \text{if } i < 0 \\ i & \text{otherwise} \end{cases}.$$

Isabelle: `libisa/arith_range.intwd_as_nat`

Definition 3.7 (Conversion from naturals to integers) A 32-bit natural number n is converted into an integer number by means of the function

$$n2i(n) \stackrel{\text{def}}{=} \begin{cases} n - 2^{32} & \text{if } 2^{31} \leq n < 2^{32} \\ n & \text{otherwise} \end{cases}.$$

Isabelle: `libisa/arith_range.natwd_as_int`

Register Files

Our observation shows that in Verisoft programs use integers more frequently. Therefore, we decided to represent all registers except the program counters with integers. Register files are represented therefore as lists of integers:

$$Regf_{\text{ASM}} \stackrel{\text{def}}{=} \mathbb{Z}^*.$$

For a register file $r :: Regf_{\text{ASM}}$ we define the well-formedness predicate that demands the file length to be 32 and for each its item to be a VAMP assembly integer:

$$Regf_{\text{ASM}}\surd(r) \stackrel{\text{def}}{=} |r| = 32 \ \wedge \ \forall\, i < 32 : \ \mathbb{Z}_{32}\surd(r[i]).$$

Memories

On the assembly level we represent memories as mappings from naturals to integers:

$$Mem_{\text{ASM}} \stackrel{\text{def}}{=} \mathbb{N} \mapsto \mathbb{Z}.$$

We suppose the memory to be word-addressable and thus each memory cell must represented with a VAMP assembly integer. In order to support this we define the following read and write functions.

Definition 3.8 (Assembly memory read) Reading from an assembly memory $m :: Mem_{\mathrm{ASM}}$ at an address $a :: \mathbb{Z}$ is done with the function

$$m_{word}(a) \stackrel{\text{def}}{=} m(a/4).$$

Isabelle: `VAMPasm/Memory.data_mem_read`

Definition 3.9 (Assembly memory write) Writing to assembly memory $m :: Mem_{\mathrm{ASM}}$ at an address $a :: \mathbb{Z}$ is done with the function

$$\mathit{mem\text{-}update}_{\mathrm{ASM}}(m, a, \mathit{data}) = m',$$

such that

$$m'(i) = \begin{cases} \mathit{data} & \text{if } i = a/4 \\ m(i) & \text{otherwise} \end{cases}.$$

Isabelle: `VAMPasm/Memory.data_mem_write`

Additionally we define a well-formedness predicate over an assembly memory $m :: Mem_{\mathrm{ASM}}$:

$$Mem_{\mathrm{ASM}}\surd(m) \stackrel{\text{def}}{=} \forall\, a :\, \mathbb{N}_{32}\surd(a) \longrightarrow \mathbb{Z}_{32}\surd(m_{word}(a)).$$

3.2.3 Configurations

Configurations c_{ASM} of an assembly machine are modeled with the record C_{ASM} which has the following fields:

- the program counter $pc :: \mathbb{N}$ and the delayed program counter $dpc :: \mathbb{N}$,
- the general purpose register file $gpr :: Regf_{\mathrm{ASM}}$ and the special purpose register file $spr :: Regf_{\mathrm{ASM}}$, and
- the memory $m :: Mem_{\mathrm{ASM}}$.

VAMP assembly configurations are also called *VAMP assembly machines*.

We define functions for reading both general and special purpose register files of the VAMP assembly model in much the same way we define them for VAMP ISA (cf. Definitions 3.1 and 3.2).

Definition 3.10 (Assembly GPR read) Reading from an assembly general purpose register file $r :: Regf_{\mathrm{ASM}}$ at an index $i :: \mathbb{N}$ is done with the function

$$\mathit{gpr\text{-}read}_{\mathrm{ASM}}(r, i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } i = 0 \\ r[i] & \text{otherwise} \end{cases}.$$

Isabelle: `VAMPasm/Config.reg`

Table 3.1 defines the set $sprs_{\mathrm{ASM}} :: 2^{\mathbb{N}}$ of special purpose register decimal indices which we use in VAMP assembly model.

Definition 3.11 (Assembly SPR read) Reading from an assembly special purpose register file $r :: \mathit{Regf}_{\mathrm{ASM}}$ at an index $i :: \mathbb{N}$ is done with the function

$$\mathit{spr\text{-}read}_{\mathrm{ASM}}(r, i) \stackrel{\mathrm{def}}{=} \begin{cases} r[i] & \text{if } i \in \mathit{sprs}_{\mathrm{ASM}} \\ 0 & \text{otherwise} \end{cases}.$$

Isabelle: `VAMPasm/Config.sreg`

The well-formedness requirements over VAMP assembly configurations are stated by means of the following predicate.

Definition 3.12 (Valid assembly configuration) Let c_{ASM} be an assembly model configuration. The predicate

$$\begin{aligned} \mathit{asm}\surd(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=}\ & \mathbb{N}_{32}\surd(c_{\mathrm{ASM}}.\mathit{pc}) \\ \wedge\ & \mathbb{N}_{32}\surd(c_{\mathrm{ASM}}.\mathit{dpc}) \\ \wedge\ & \mathit{Regf}_{\mathrm{ASM}}\surd(c_{\mathrm{ASM}}.\mathit{gpr}) \\ \wedge\ & \mathit{Regf}_{\mathrm{ASM}}\surd(c_{\mathrm{ASM}}.\mathit{spr}) \\ \wedge\ & \mathit{Mem}_{\mathrm{ASM}}\surd(c_{\mathrm{ASM}}.m) \end{aligned}$$

states the validity requirements on c_{ASM}.
Isabelle: `VAMPasm/Config.is_ASMcore`

3.2.4 Execution modes

In contrast to ISA in assembly model we do not have a mechanism to switch between the modes (since interrupts are not visible). We use the mode to allow/forbid the special move instructions. The system mode definition follows the one from the ISA model.

$$\mathit{is\text{-}sys\text{-}mode}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} c_{\mathrm{ASM}}.\mathit{spr}[\mathit{mode}] = 0.$$

A further distinctive feature of system-mode executions is that all interrupts are masked out, denoted by an empty status register:

$$\mathit{is\text{-}sys\text{-}exec}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} c_{\mathrm{ASM}}.\mathit{spr}[\mathit{mode}] = 0 \ \wedge\ c_{\mathrm{ASM}}.\mathit{spr}[\mathit{sr}] = 0.$$

3.2.5 Instructions

We model VAMP instructions on the assembly level with the inductive data type *Instr*. Its constructors have names of instruction mnemonics from Table 3.2 Since register and immediate fields of an instruction *instr* are formally modeled by unbounded numbers, we introduce the instruction validity predicate

$$\mathit{instr}\surd :: \mathit{Instr} \mapsto \mathbb{B}$$

which bounds the fields to the appropriate lengths. We omit the formal definition here because is straightforward and rather voluminous — the reader can consult the definition `VAMPasm/Instr.is_instr` in Isabelle/HOL.

Parameters to the constructor of an instruction *instr* like source and destination registers or an immediate constant are obtained like $rs_1(instr)$, $rd(instr)$, $imm(instr)$, etc.

Since memory cells are modeled as integers we need a way to convert integers to instructions. This is done by the function

$$\textit{int-to-instr} :: \mathbb{Z} \mapsto \textit{Instr}.$$

First, this function decomposes its argument into the opcode, register, and immediate fields. Then it makes a case split on the opcode, determines the appropriate constructor of the data type *Instr*, and supplements it with register and immediate parts. However, this simple idea is implemented with tens of lines in Isabelle/HOL in the definition `VAMPasm/Instr_convert.int_to_instr`.

Since it is not possible to convert every integer to an instruction we use the predicate

$$\textit{decodable} :: \mathbb{Z} \mapsto \mathbb{B},$$

which tests whether an integer could be converted to an instruction.

Additionally, we define the function

$$\textit{instr-to-int} :: \textit{Instr} \mapsto \mathbb{Z}$$

to perform the conversion in other direction.

The current instruction to be executed in an assembly configuration c_{ASM} is obtained through the function

$$\textit{instr}(c_{\text{ASM}}) \stackrel{\text{def}}{=} \textit{int-to-instr}(c_{\text{ASM}}.m_{\textit{word}}(c_{\text{ASM}}.\textit{dpc})).$$

As in ISA, we introduce the predicates over instructions to determine what instruction or group of instructions it corresponds to. It is done for each instruction constructor, e.g., *is-instr-trap*, or for a group, like *is-ls* for all memory access instructions, or *is-store* only for memory write access. We distinguish also between memory accesses of different width: *is-ls-w* for the whole word, or *is-ls-hw* only for the half.

3.2.6 Semantics

The effect of a single instruction execution on the assembly configuration is defined by the function

$$\textit{exec}_{\text{instr}} :: C_{\text{ASM}} \times \textit{Instr} \mapsto C_{\text{ASM}}.$$

The function $\textit{exec}_{\text{instr}}(c_{\text{ASM}}, \textit{instr})$ is defined in Isabelle/HOL under `VAMPasm/Exec.exec_instr` by structural induction on the constructor type of *instr* and yields an updated assembly configuration c'_{ASM}. Definitions of each induction case use a number of common auxiliary functions. Next, we define these auxiliary functions for (i) arithmetic, logical, shift, and constant load operations — we call this group arithmetic instructions, (ii) test and set operation — this group is named comparison instructions, (iii) instructions for load, and (iv) store instructions. All functions from these group increment the program counter by 4 and hence their effect comprises:

$$\begin{aligned} c'_{\text{ASM}}.\textit{dpc} &= c_{\text{ASM}}.\textit{pc}, \\ c'_{\text{ASM}}.\textit{pc} &= c_{\text{ASM}}.\textit{pc} +_{32} 4. \end{aligned}$$

Arithmetic, comparison and load instruction change the general purpose register file of the assembly configuration while store instructions change its memory. All other components of c'_{ASM} stay equal to those of c_{ASM}.

Arithmetic Instructions

Execution of arithmetic instructions is modeled by the function

$$exec_{\text{arith}} :: C_{\text{ASM}} \times (\mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z}) \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \mapsto C_{\text{ASM}},$$

$$exec_{\text{arith}}(c_{\text{ASM}}, f, op_1, op_2, dst) \;=\; c'_{\text{ASM}},$$

which writes the result of the application of the arithmetic function f to operands op_1 and op_2 into the general purpose register with index dst:

$$c'_{\text{ASM}}.gpr[i] \;=\; \begin{cases} f(op_1, op_2) & \text{if } i = dst \\ c_{\text{ASM}}.gpr[i] & \text{otherwise} \end{cases}.$$

In the induction cases of the definition of $exec_{\text{instr}}$ where we use $exec_{\text{arith}}$ we substitute an appropriate arithmetic operation for f, e.g., the case of the definition for the addition instruction `add` is as follows:

$$\begin{aligned} exec_{\text{instr}}(c_{\text{ASM}}, \texttt{add}(rd, rs_1, rs_2)) \stackrel{\text{def}}{=}\; & exec_{\text{arith}}(c_{\text{ASM}}, \\ & \textit{int-plus}, \\ & \textit{gpr-read}_{\text{ASM}}(c_{\text{ASM}}.gpr, rs_1), \\ & \textit{gpr-read}_{\text{ASM}}(c_{\text{ASM}}.gpr, rs_2), \\ & rd), \end{aligned}$$

with $\textit{int-plus}(x, y) = \textit{n2i}(\textit{i2n}(x) +_{32} \textit{i2n}(y))$.

Comparison Instructions

Execution of comparison instructions is defined by the function

$$exec_{\text{comp}} :: C_{\text{ASM}} \times (\mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{B}) \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \mapsto C_{\text{ASM}},$$

$$exec_{\text{comp}}(c_{\text{ASM}}, \stackrel{?}{\circ}, op_1, op_2, dst) = c'_{\text{ASM}},$$

which writes one to the destination general purpose register in case $op_1 \stackrel{?}{\circ} op_2$ evaluates to true and zero otherwise:

$$c'_{\text{ASM}}.gpr[i] \;=\; \begin{cases} 1 & \text{if } op_1 \stackrel{?}{\circ} op_2 \;\wedge\; i = dst \\ 0 & \text{if } \neg(op_1 \stackrel{?}{\circ} op_2) \;\wedge\; i = dst \\ c_{\text{ASM}}.gpr[i] & \text{otherwise} \end{cases}.$$

An appropriate comparison predicate is substituted for $p?$ in the induction cases of the $exec_{\text{instr}}$ definition where we use $exec_{\text{comp}}$. For instance the induction case for the equality test performed by the instruction `seq` is defined as:

$$\begin{aligned} exec_{\text{instr}}(c_{\text{ASM}}, \texttt{seq}(rd, rs_1, rs_2)) \stackrel{\text{def}}{=}\; & exec_{\text{comp}}(c_{\text{ASM}}, \\ & =, \\ & \textit{gpr-read}_{\text{ASM}}(c_{\text{ASM}}.gpr, rs_1), \\ & \textit{gpr-read}_{\text{ASM}}(c_{\text{ASM}}.gpr, rs_2), \\ & rd). \end{aligned}$$

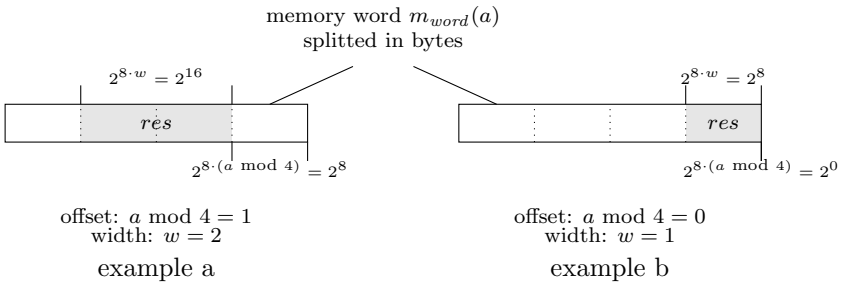

Figure 3.2: Examples of conversions for data load.

Load Instructions

In order to model load instruction we define the following function:

$$exec_{\text{load}} :: C_{\text{ASM}} \times (Bv \mapsto Bv) \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto C_{\text{ASM}},$$

$$exec_{\text{load}}(c_{\text{ASM}}, xtf, a, w, dst) \;=\; c'_{\text{ASM}}.$$

It writes the content of the memory at the address a into the general purpose register indexed by *dst*. The function is parametrized by (i) the width of memory access w which can be either 1, 2, or 4 corresponding to byte, half-word, and word access, and (ii) the sign-extension function *xtf* which is subject to instantiation either by *sxt* or *fill0* representing signed and unsigned memory operations. We formalize all kinds of load operations to assembly memory with the following function.

Definition 3.13 (Assembly load) Reading of w bytes from an assembly memory m at an address a with respect to a sign-extension function *xtf* is done by means of the function

$$load_{\text{ASM}} :: Mem_{\text{ASM}} \times \mathbb{N} \times \mathbb{N} \times (Bv \mapsto Bv) \mapsto \mathbb{Z}.$$

First, we extract the needed data from the memory word (cf. Figure 3.2)

$$res \;=\; \frac{i2n(m_{word}(a))}{2^{8\cdot(a \bmod 4)}} \bmod 2^{8\cdot w}.$$

Further, we extend it with the given function, preliminary adding the leading zeros up to the length $8 \cdot w$:

$$load_{\text{ASM}}(m, a, w, xtf) \;\stackrel{\text{def}}{=}\; [xtf(0^{8\cdot w - |bin(res)|} \circ bin(res))]$$

Isabelle: `VAMPasm/Exec.load_from_mem`

Our motivation to do the extension of the result twice (first with zeros and then with the given extension function) is that the binary representation $bin(res)$ is a bit vector of minimal length which is enough to encode the number. The length of $bin(res)$ is possibly less than $8 \cdot w$ and the most significant bit is always 1. Hence, the sign extension of the binary representation might lead to a result which differs from the one obtained by performing analogous manipulations on bit vectors.

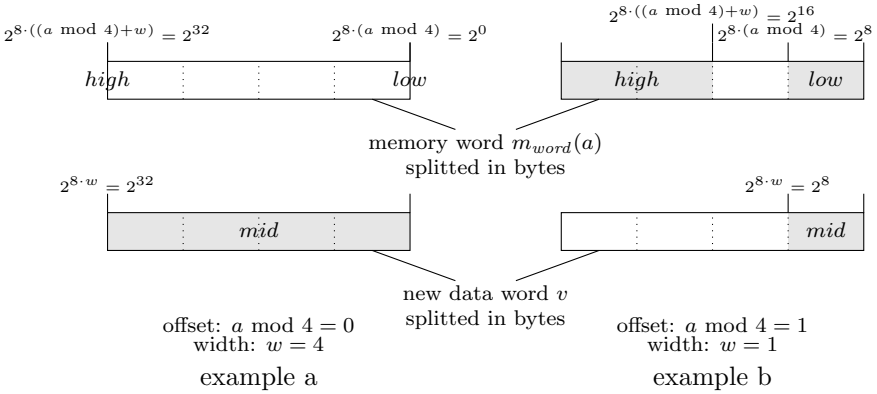

Figure 3.3: Examples conversions for data store.

In the formal definition of $exec_{\text{load}}$ we proceed with memory read, and write result into the register file:

$$c'_{\text{ASM}}.gpr[i] \;=\; \begin{cases} load_{\text{ASM}}(c_{\text{ASM}}.m, a, w, xtf) & \text{if } i = dst \\ c_{\text{ASM}}.gpr[i] & \text{otherwise} \end{cases}.$$

Having this definition we can specify, for instance, the induction case of $exec_{\text{instr}}$ for loading a word from the memory by means of the instruction `lw`:

$$exec_{\text{instr}}(c_{\text{ASM}}, \texttt{lw}(rd, rs, imm)) \stackrel{\text{def}}{=} exec_{\text{load}}(c_{\text{ASM}}, sxt, \textit{ls-target}(c_{\text{ASM}}), 4, rd).$$

In general, the case of "4" is computed by the function $\textit{ls-width}(c_{\text{ASM}})$ for all load and store instructions. The function *ls-target* computes effective address for memory operations (load and store) under the condition that the current instruction is a memory access instruction with the source register *rs* and immediate constant *imm*:

$$\textit{ls-target}(c_{\text{ASM}}) \stackrel{\text{def}}{=} \\ i2n(\textit{gpr-read}_{\text{ASM}}(c_{\text{ASM}}.gpr, rs(\textit{instr}(c_{\text{ASM}})))) +_{32} i2n(imm(\textit{instr}(c_{\text{ASM}}))).$$

Store Instructions

Semantics of store instructions is defined by the function

$$exec_{\text{store}} :: C_{\text{ASM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto C_{\text{ASM}},$$

$$exec_{\text{store}}(c_{\text{ASM}}, a, w, src) \;=\; c'_{\text{ASM}},$$

which stores w bytes of the data from a general purpose register *src* in the memory of the assembly machine at the address a. Likewise load operations, we formalize store operations with the following definition.

Definition 3.14 (Assembly store) Writing of w lowest bytes from a value v into an assembly memory m at an address a is done by means of the function

$$store_{\text{ASM}} :: Mem_{\text{ASM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \mapsto Mem_{\text{ASM}}.$$

In the formal definition we first compute the part of the number to be written

$$mid \;=\; (i2n(v) \bmod 2^{8\cdot w}) \cdot 2^{8\cdot(a \bmod 4)}.$$

Then, we read the value from the memory at the address we are writing to and cut the lower and higher parts that should be ignored (cf. Figure 3.3):

$$\begin{aligned} high &= \frac{i2n(m_{word}(a))}{2^{8\cdot((a \bmod 4)+w)}} \cdot 2^{8\cdot((a \bmod 4)+w)}, \\ low &= i2n(m_{word}(a)) \bmod 2^{8\cdot(a \bmod 4)}. \end{aligned}$$

At the end we combine all three parts and write the result into the memory:

$$store_{\mathrm{ASM}}(m, a, w, v) \stackrel{\mathrm{def}}{=} mem\text{-}update_{\mathrm{ASM}}(m, a, n2i(high +_{32} mid +_{32} low)).$$

Isabelle: `VAMPasm/Exec.store_to_mem`

The updated assembly configuration c'_{ASM} returned by the function $exec_{\mathrm{store}}$ comprises the memory defined as follows:

$$c'_{\mathrm{ASM}}.m \;=\; store_{\mathrm{ASM}}(c_{\mathrm{ASM}}.m, a, w, gpr\text{-}read_{\mathrm{ASM}}(c_{\mathrm{ASM}}.gpr, src)).$$

After all, induction cases for store instruction in the definition of $exec_{\mathrm{instr}}$ make use of $exec_{\mathrm{store}}$. For instance, effect of the store byte instruction `sb` is given as

$$exec_{\mathrm{instr}}(c_{\mathrm{ASM}}, \mathtt{sb}(rd, rs, imm)) \stackrel{\mathrm{def}}{=} exec_{\mathrm{store}}(c_{\mathrm{ASM}}, ls\text{-}target(c_{\mathrm{ASM}}), 1, rd).$$

Special and Control Instructions

Semantics of the remaining VAMP instructions — special and control instructions — is defined directly in $exec_{\mathrm{instr}}$.

Instruction for exchanging values of special and general purpose registers `movi2s` and `movs2i` are allowed only in the system mode. Then, the semantics of these instructions is defined as follows[2]:

$$exec_{\mathrm{instr}}(c_{\mathrm{ISA}}, \mathtt{movs2i}(rd, sa)) \stackrel{\mathrm{def}}{=} \begin{cases} c_{\mathrm{ISA}} & \text{if } \neg is\text{-}sys\text{-}mode_{\mathrm{ASM}}(c_{\mathrm{ISA}}) \\ c'_{\mathrm{ISA}} & \text{otherwise} \end{cases},$$

where program counters are incremented in the same way as described for all previous instructions and

$$c'_{\mathrm{ASM}}.gpr[i] \;=\; \begin{cases} spr\text{-}read_{\mathrm{ASM}}(c_{\mathrm{ASM}}.spr[sa]) & \text{if } i = rd \\ c_{\mathrm{ASM}}.gpr[i] & \text{otherwise} \end{cases}.$$

As `movi2s` transfers values in opposite direction, i.e., from general to special purpose registers, its definition simply swaps *gpr* and *spr*.

The `trap` instruction only increments the program counter by four and stores its old value in the delayed program counter. The `rfe` instruction does not affect the VAMP assembly configuration.

[2] In Isabelle/HOL theories indices of special purpose registers for integration of a floating point unit are defined: the rounding mode register *RM*, the IEEE flags register *IEEEf*:, and the floating point condition code *FCC*. It is allowed to access these registers in user mode.

The jump instruction `j` saves the program counter into the delayed one and adds an immediate to *pc*:

$$exec_{\text{instr}}(c_{\text{ASM}}, \texttt{j}(imm)) \stackrel{\text{def}}{=} c'_{\text{ASM}},$$

$$\begin{aligned} c'_{\text{ASM}}.dpc &= c_{\text{ASM}}.pc, \\ c'_{\text{ASM}}.pc &= c_{\text{ASM}}.pc +_{32} i2n(imm). \end{aligned}$$

The jump-and-link instruction `jal` additionally saves the program counter value increased by four in the general purpose register 32: the formal definition of

$$exec_{\text{instr}}(c_{\text{ASM}}, \texttt{jal}(imm)) \stackrel{\text{def}}{=} c'_{\text{ASM}}$$

includes

$$c'_{\text{ASM}}.gpr[i] = \begin{cases} n2i(c_{\text{ASM}}.pc +_{32} 4) & \text{if } i = 31 \\ c_{\text{ASM}}.gpr[i] & \text{otherwise} \end{cases}.$$

Semantics of jump register `jr` and jump-and-link register instructions `jar` coincide with those of `j` and `jal`, respectively, with the only difference that the program counter is assigned the value of general purpose register at index *rs*, a parameter to `jr` and `jar`:

$$c'_{\text{ASM}}.pc = i2n(gpr\text{-}read_{\text{ASM}}(c_{\text{ASM}}.gpr, rs)).$$

The branch-on-zero instruction `beqz` takes a look in the source general purpose register and adds an immediate constant to the program counter in case the former stores a zero value, or simply 4 otherwise:

$$exec_{\text{instr}}(c_{\text{ASM}}, \texttt{beqz}(rs, imm)) \stackrel{\text{def}}{=} c'_{\text{ASM}},$$

$$\begin{aligned} c'_{\text{ASM}}.dpc &= c_{\text{ASM}}.pc, \\ c'_{\text{ASM}}.pc &= \begin{cases} c_{\text{ASM}}.pc +_{32} i2n(imm) & \text{if } gpr\text{-}read_{\text{ASM}}(c_{\text{ASM}}.gpr, rs) \\ c_{\text{ASM}}.pc +_{32} 4 & \text{otherwise} \end{cases}. \end{aligned}$$

The definition of the branch-on-non-zero instruction `bnez` only swaps cases in the case distinction.

The formal definition of $exec_{\text{instr}}$ in Isabelle/HOL is `VAMPasm/Exec.exec_instr`.

Transition Function

Executions of the assembly model are modeled by the transition function $\delta_{\text{ASM}} :: C_{\text{ASM}} \mapsto C_{\text{ASM}}$. The function $\delta_{\text{ASM}}(c_{\text{ASM}})$ executes the current instruction with the function $exec_{\text{instr}}$:

$$\delta_{\text{ASM}}(c_{\text{ASM}}) \stackrel{\text{def}}{=} exec_{\text{instr}}(c_{\text{ASM}}, instr(c_{\text{ASM}})).$$

Several steps of the assembly machine are done by the function δ^n_{ASM}, which is defined by induction on n:

$$\begin{aligned} \delta^0_{\text{ASM}}(c_{\text{ASM}}) &= c_{\text{ASM}}, \\ \delta^{n+1}_{\text{ASM}}(c_{\text{ASM}}) &= \delta^n_{\text{ASM}}(\delta_{\text{ASM}}(c_{\text{ASM}})). \end{aligned}$$

3.2.7 Assembly Programs

Assembly semantics models computations of assembly programs. We represent them as lists of assembly instructions: an assembly program is an instance of the type

$$\Pi_{\text{ASM}} \stackrel{\text{def}}{=} \mathit{Instr}^*.$$

An assembly memory region can be interpreted as an assembly program. Let us denote the list of n integers which resides in an assembly memory m starting at an address a by

$$\mathit{get\text{-}data}(m, a, \mathit{len}) \stackrel{\text{def}}{=} [m(a/4), m(a/4+1), \ldots, m(a/4+n-1)].$$

Having this notation, we define the function that retrieves an assembly program from the memory.

Definition 3.15 (Retrieving an assembly program) The program of the length $\mathit{len} :: \mathbb{N}$ is retrieved from an assembly memory $m :: \mathit{Mem}_{\text{ASM}}$ starting from an address $a :: \mathbb{N}$ by means of the function

$$\mathit{get\text{-}\pi}(m, a, \mathit{len}) \stackrel{\text{def}}{=} \pi$$

with

$$\pi[i] \;=\; \mathit{int\text{-}to\text{-}instr}(\mathit{get\text{-}data}(m, a, \mathit{len})[i]).$$

Isabelle: `VAMPasm/Memory.get_instr_list`

We call a region of the memory *decodable* if values of its each memory cell could be converted to instructions:

$$\mathit{decodable\text{-}\pi}(\mathit{data}) \stackrel{\text{def}}{=} \forall\, i < |\mathit{data}| :\; \mathit{decodable}(\mathit{data}[i]).$$

Lemma 3.16 (Validity of assembly programs) Instructions obtained in a decodable region from a valid memory are valid:

$$\begin{aligned} &\mathit{Mem}_{\text{ASM}}\surd(m) \wedge \mathit{decodable\text{-}\pi}(\mathit{get\text{-}data}(m, a, \mathit{len})) \\ &\qquad\longrightarrow\; \forall\, i < \mathit{len} :\; \mathit{instr}\surd(\mathit{get\text{-}\pi}(m, a, \mathit{len})[i]). \end{aligned}$$

Isabelle: `VAMPasm/Instr_convert.decodable_imp_is_instr`

3.3 Devices

This section introduces the devices model used in the thesis. We start by sketching a generic device model and later illustrate by the hard disk example how this model can be instantiated with concrete devices. We show how several devices are organized in a devices system. The reader should consult [64] for additional information on the devices model in general, and [51] for a hard disk.

In Isabelle devices have outputs to the external environment. However, they are irrelevant to the current work. Therefore, we omit these outputs throughout the thesis.

3.3.1 Device Model

A device of type x is modeled as a finite transition system with configurations $c_x :: C_x$ and a transition function δ_x. The step function takes the current state of the device $c_x :: C_x$, an input $mifi_t :: Mifi_t$ from the memory interface of the processor, and an input $eifi_x :: Eifi_x$ from the external interface of a non-modeled external environment. It returns a devices' updated state $c'_x :: C_x$ and an output $mifo_t :: Mifo_t$ to the memory interface of the processor. Thus, the transition function has the following signature:

$$\delta_x :: C_x \times Mifi_t \times Eifi_x \mapsto C_x \times Mifo_t.$$

The sets of inputs $Eifi_x$ from the external environment are device-specific, while the memory interfaces $Mifi_t$ and $Mifo_t$ depend only on the type t they are represented with. Devices are accessed via (at most) 1024 word-sized ports.

Memory Interface

A memory interface between a processor and devices is specified by the memory interface inputs $mifi_t$ and the outputs $mifo_t$. The inputs are given by a processor to a device, and the outputs are produced by a device for a processor.

The memory interface inputs $mifi_t$ are represented by a record of type $Mifi_t$ with fields representation depending on the type t of the memory interface. The individual fields of the record are:

- the read flag $mifi_t.rd$, which indicates a read operation on a device,
- the write flag $mifi_t.wr$, which indicates a write operation on a device,
- the access address $mifi_t.a$, and
- the word-sized data input $mifi_t.din$ used for write accesses to a device.

We deal with two representations of the memory interface inputs. The natural representation $mifi_{\mathbb{N}} :: Mifi_{\mathbb{N}}$ uses boolean values to represent the flags $mifi_{\mathbb{N}}.rd, mifi_{\mathbb{N}}.wr :: \mathbb{B}$, and naturals for the addresses and data input components $mifi_{\mathbb{N}}.a, mifi_{\mathbb{N}}.din :: \mathbb{N}$. In the bit-vector representation $mifi_{Bv} :: Mifi_{Bv}$ the flags are bits $mifi_{Bv}.rd, mifi_{Bv}.wr :: Bit$, and the addresses and data inputs are bit vectors $mifi_{Bv}.a, mifi_{Bv}.din :: Bv$.

The following constants denote the idle memory interface input:

$$\begin{aligned} \varepsilon\text{-}mifi_{\mathbb{N}} &\stackrel{\text{def}}{=} (\mathsf{F}, \mathsf{F}, \mathsf{A}, \mathsf{A}) :: Mifi_{\mathbb{N}}, \\ \varepsilon\text{-}mifi_{Bv} &\stackrel{\text{def}}{=} (0, 0, \mathsf{A}, \mathsf{A}) :: Mifi_{Bv}. \end{aligned}$$

The conversion from the bit-vector to the natural representation of memory interface inputs is done by the function

$$mifi\text{-}bv\text{-}to\text{-}nat :: Mifi_{Bv} \mapsto Mifi_{\mathbb{N}}$$

which yields the natural memory interface inputs $mifi_{\mathbb{N}} = mifi\text{-}bv\text{-}to\text{-}nat(mifi_{Bv})$:

$$\begin{aligned} mifi_{\mathbb{N}}.rd &= (mifi_{Bv}.rd = 1), \\ mifi_{\mathbb{N}}.wr &= (mifi_{Bv}.wr = 0), \\ mifi_{\mathbb{N}}.a &= \langle mifi_{Bv}.a \rangle, \\ mifi_{\mathbb{N}}.din &= \langle mifi_{Bv}.din \rangle. \end{aligned}$$

Table 3.4: Concrete devices and their aliases

Device name	Alias
ATA/ATAPI hard disk	HD
Timer	TIMER
Automotive bus controller	ABC
Network interface card (NE2000)	NIC
Serial interface (UART16550A)	UART

The memory interface output is a singleton, represented either as a natural $\mathit{mifo}_{\mathbb{N}} :: \mathit{Mifo}_{\mathbb{N}}$ or a bit vector $\mathit{mifo}_{Bv} :: \mathit{Mifo}_{Bv}$. We declare the constants $\varepsilon\text{-}\mathit{mifo}_{\mathbb{N}} :: \mathit{Mifo}_{\mathbb{N}}$ and $\varepsilon\text{-}\mathit{mifo}_{Bv} :: \mathit{Mifo}_{Bv}$ to denote the idle memory interface output.

3.3.2 Concrete Devices

A device model can be instantiated with particular devices. In CVM we use five explicit formal models of concrete devices. Table 3.4 depicts their names and aliases. Aliases are written as subscript after an entity to denote that the entity is related to a particular device. For example, $c_{\text{TIMER}} :: C_{\text{TIMER}}$ is a configuration of the timer, δ_{TIMER} is its transition function, and $\mathit{eifi}_{\text{TIMER}} :: \mathit{Eifi}_{\text{TIMER}}$ is timer's inputs from the external environment, respectively. Since all devices are treated in exactly the same fashion we will use only the model of a hard disk within this thesis. The details of the automotive bus controller model are described in [6]. For the serial interface cf. [3]. The detailed description of the hard-disk model can be found in [51].

Hard Disk

Next, we sketch details of the hard-disk model relevant for the thesis. We use the model of the disk based on the ATA/ATAPI protocol. The hard disk is parameterized over the number of sectors it has. Each sector has a size of `WORDS_PER_SECTOR` $= 128$ words. The processor can issue read or write commands to a range of sectors, by writing the start address and the count of sectors to a special port. Each sector is then read/written word by word from/to a sector buffer. After a complete sector is read/written from/to the sector buffer, the hard disk needs some time to transfer data to the sector memory. This amount of time is modeled as non-determinism by an oracle input from the external environment $\mathit{Eifi}_{\text{HD}} \stackrel{\text{def}}{=} \{1, 0\}$. The value $\mathit{eifi}_{hd} = 1$ indicates the end of the transfer.

Hard disk configurations c_{HD} are represented by a record of type C_{HD}. The record comprises fields for modeling hard-disk internal functionality as well as contents stored on the hard disk. For this thesis we are interested only in the three following fields of the hard-disk record:

- the number of sectors $c_{\text{HD}}.s :: \mathbb{N}$ which has to be less than or equal to `MAX_SECTORS` $= 2^{28}$,
- the swap memory $c_{\text{HD}}.sm :: \mathbb{N}^*$ which represents the hard-disk content as a list of natural numbers, and
- the control state $c_{\text{HD}}.cs :: \{\texttt{HD\_IDLE}, \texttt{HD\_BRD}, \texttt{HD\_BWR}, \texttt{HD\_PRD}, \texttt{HD\_PWR}, \texttt{HD\_ERR}\}$.

In the state HD_IDLE the disk is ready to process new commands. Reading from the disk starts in the state HD_BRD by filling the disk buffer which is then read by the processor when the disk is in the state HD_PRD. Similarly, write commands visit the states HD_PWR and HD_BWR. In case an invalid command is issued, the disk transits to the error state HD_ERR.

The well-formedness predicate over the hard disk state $hd\surd(c_{\mathrm{HD}})$ requires, among others, that the length of the swap memory is defined by the number of sectors:

$$|c_{\mathrm{HD}}.sm| \;=\; c_{\mathrm{HD}}.s \cdot \texttt{WORDS\_PER\_SECTOR},$$

and that each cell of the swap memory is a valid VAMP assembly natural number:

$$\forall\, i < |c_{\mathrm{HD}}.sm| :\; \mathbb{N}_{32}\surd(c_{\mathrm{HD}}.sm[i]).$$

3.3.3 Generalized Devices

The concept of generalized devices allows us to deal with devices in a generic fashion, i.e., without having knowledge about particular kinds of devices. Generalized devices are represented by inductive data types, where each inductive constructor corresponds to a certain device. As mentioned before we consider only the hard disk in the thesis, however the concept is easily expendable to all devices from Table 3.4. To stress that we will also refer to the timer in the definitions below.

Generalized Configurations

The generalized device configuration c_{GD} is defined by the following inductive data type:

$$\begin{array}{rcl} c_{\mathrm{GD}} :: C_{\mathrm{GD}} & \stackrel{\mathrm{def}}{=} & \mathit{dev\text{-}hd}(C_{\mathrm{HD}}) \\ & | & \mathit{dev\text{-}timer}(C_{\mathrm{TIMER}}) \\ & | & \ldots \\ & | & \mathit{idle\text{-}dev}. \end{array}$$

The constructor *idle-dev* is used to model an idle device which is used among others to model an illegal device access.

In order to determine whether a generalized device configuration c_{GD} corresponds to a particular device configuration we use the following predicates:

$$\begin{array}{rcl} \mathit{is\text{-}dev\text{-}hd}(c_{\mathrm{GD}}) & = & \exists\, c_{\mathrm{HD}} :\; c_{\mathrm{GD}} = \mathit{dev\text{-}hd}(c_{\mathrm{HD}}), \\ \mathit{is\text{-}dev\text{-}timer}(c_{\mathrm{GD}}) & = & \exists\, c_{\mathrm{TIMER}} :\; c_{\mathrm{GD}} = \mathit{dev\text{-}timer}(c_{\mathrm{TIMER}}), \\ & \ldots & \\ \mathit{is\text{-}idle\text{-}dev}(c_{\mathrm{GD}}) & = & c_{\mathrm{GD}} = \mathit{idle\text{-}dev}. \end{array}$$

To extract a particular device configuration from a generalized device configuration we use a function of the same name as the device extended with a prefix *the-*, e.g., for the hard disk:

$$\mathit{the\text{-}hd}(\mathit{dev\text{-}hd}(c_{\mathrm{HD}})) \stackrel{\mathrm{def}}{=} c_{\mathrm{HD}}.$$

Generalized External Inputs

In the same fashion we define the generalized inputs from the external environment:

$$\begin{array}{rcl} \mathit{eifi}_{\mathrm{GD}} :: \mathit{Eifi}_{\mathrm{GD}} & \stackrel{\mathrm{def}}{=} & \mathit{eifi\text{-}hd}(\mathit{Eifi}_{\mathrm{HD}}) \\ & | & \mathit{eifi\text{-}timer}(\mathit{Eifi}_{\mathrm{TIMER}}) \\ & | & \ldots \\ & | & \mathit{idle\text{-}eifi}. \end{array}$$

The constructor *idle-eifi* is supposed to use at places where we speak about idle device *idle-dev*.

The following predicates test whether a generalized input $\mathit{eifi}_{\mathrm{GD}}$ correspond to a particular device input:

$$\begin{array}{rcl} \mathit{is\text{-}eifi\text{-}hd}(\mathit{eifi}_{\mathrm{GD}}) & \stackrel{\mathrm{def}}{=} & \exists\ \mathit{eifi}_{\mathrm{HD}} :\ \mathit{eifi}_{\mathrm{GD}} = \mathit{eifi\text{-}hd}(\mathit{eifi}_{\mathrm{HD}}), \\ \mathit{is\text{-}eifi\text{-}timer}(\mathit{eifi}_{\mathrm{GD}}) & \stackrel{\mathrm{def}}{=} & \exists\ \mathit{eifi}_{\mathrm{TIMER}} :\ \mathit{eifi}_{\mathrm{GD}} = \mathit{eifi\text{-}timer}(\mathit{eifi}_{\mathrm{TIMER}}), \\ & \ldots & \\ \mathit{is\text{-}idle\text{-}eifi}(\mathit{eifi}_{\mathrm{GD}}) & \stackrel{\mathrm{def}}{=} & \mathit{eifi}_{\mathrm{GD}} = \mathit{idle\text{-}eifi}. \end{array}$$

Having a generalized device configuration c_{GD} and a generalized input $\mathit{eifi}_{\mathrm{GD}}$ from the external environment we need to test whether the input is compatible with the device. The following predicate is used for that:

$$\begin{array}{rcl} \mathit{is\text{-}eifi\text{-}match\text{-}dev}(c_{\mathrm{GD}}, \mathit{eifi}_{\mathrm{GD}}) & \stackrel{\mathrm{def}}{=} & \mathit{is\text{-}dev\text{-}hd}(c_{\mathrm{GD}})\ \wedge\ \mathit{is\text{-}eifi\text{-}hd}(\mathit{eifi}_{\mathrm{GD}}) \\ & \vee & \mathit{is\text{-}dev\text{-}timer}(c_{\mathrm{GD}})\ \wedge\ \mathit{is\text{-}eifi\text{-}timer}(\mathit{eifi}_{\mathrm{GD}}) \\ & \vee & \ldots \\ & \vee & \mathit{is\text{-}idle\text{-}dev}(c_{\mathrm{GD}})\ \wedge\ \mathit{is\text{-}idle\text{-}eifi}(\mathit{eifi}_{\mathrm{GD}}). \end{array}$$

Next, we define generalized idle external inputs. Let $\varepsilon\text{-}\mathit{eifi}_{\mathrm{HD}}$ be idle external inputs for a hard disk, let $\varepsilon\text{-}\mathit{eifi}_{\mathrm{TIMER}}$ be idle external inputs for a timer, and so on. The function

$$\varepsilon\text{-}\mathit{eifi} :: C_{\mathrm{GD}} \mapsto \mathit{Eifi}_{\mathrm{GD}}$$

generates the generalized idle external inputs from a generalized device configuration c_{GD}:

$$\begin{array}{rcl} \varepsilon\text{-}\mathit{eifi}(\mathit{dev\text{-}hd}(c_{\mathrm{HD}})) & \stackrel{\mathrm{def}}{=} & \mathit{eifi\text{-}hd}(\varepsilon\text{-}\mathit{eifi}_{\mathrm{HD}}), \\ \varepsilon\text{-}\mathit{eifi}(\mathit{dev\text{-}timer}(c_{\mathrm{TIMER}})) & \stackrel{\mathrm{def}}{=} & \mathit{eifi\text{-}timer}(\varepsilon\text{-}\mathit{eifi}_{\mathrm{TIMER}}), \\ & \ldots & \\ \varepsilon\text{-}\mathit{eifi}(\mathit{idle\text{-}dev}) & \stackrel{\mathrm{def}}{=} & \mathit{idle\text{-}eifi}. \end{array}$$

Generalized Transitions

The transition function δ_{GD} of a generalized device takes a generalized device configuration $c_{\mathrm{GD}} :: C_{\mathrm{GD}}$, a natural representation of the memory interface input $\mathit{mifi}_{\mathbb{N}} :: \mathit{Mifi}_{\mathbb{N}}$, and a generalized external input $\mathit{eifi}_{\mathrm{DEV}} :: \mathit{Eifi}_{\mathrm{DEV}}$ as arguments. It returns an updated generalized device configuration $c'_{\mathrm{GD}} :: C_{\mathrm{GD}}$ and a natural representation of the memory interface output $\mathit{mifo}_{\mathbb{N}} :: \mathit{Mifo}_{\mathbb{N}}$. Thus the signature of the transition function is:

$$\delta_{\mathrm{GD}} :: C_{\mathrm{GD}} \times \mathit{Mifi}_{\mathbb{N}} \times \mathit{Eifi}_{\mathrm{DEV}} \mapsto C_{\mathrm{GD}} \times \mathit{Mifo}_{\mathbb{N}}.$$

We define the generalized transition inductively on the generalized external inputs and the generalized device configuration. In case the input is compatible with the device, we apply the step function for that device. Otherwise, the idle device is returned:

$$\delta_{\mathrm{GD}}(c_{\mathrm{GD}}, \mathit{mifi}, \mathit{eifi}_{\mathrm{GD}}) \stackrel{\mathrm{def}}{=} \begin{cases} (\mathit{dev}\text{-}x(c'_{\mathrm{x}}), \mathit{mifo}) & \text{if} \quad c_{\mathrm{GD}} = \mathit{dev}\text{-}x(c_{\mathrm{x}}) \\ & \quad \wedge \quad \mathit{eifi}_{\mathrm{GD}} = \mathit{eifi}\text{-}x(\mathit{eifi}_{\mathrm{x}}) \\ & \quad \wedge \quad \delta_{\mathrm{x}}(c_{\mathrm{x}}, \mathit{mifi}, \mathit{eifi}_{\mathrm{x}}) = (c'_{\mathrm{x}}, \mathit{mifo}) \\ (\mathit{idle\text{-}dev}, \varepsilon\text{-}\mathit{mifo}_{\mathbb{N}}) & \text{otherwise} \end{cases}.$$

We will use notation *.dev* to refer to the first component of the result and *.mifo* to the second.

Interrupts

As devices may produce interrupts we define the predicate

$$\mathit{is\text{-}intr\text{-}dev} :: C_{\mathrm{GD}} \mapsto \mathbb{B}$$

which indicates if there is a pending interrupt for a generalized device configuration:

$$\begin{aligned} \mathit{is\text{-}intr\text{-}dev}(\mathit{dev\text{-}hd}(c_{\mathrm{HD}})) &\stackrel{\mathrm{def}}{=} \mathit{is\text{-}intr\text{-}hd}(c_{\mathrm{HD}}), \\ \mathit{is\text{-}intr\text{-}dev}(\mathit{dev\text{-}timer}(c_{\mathrm{TIMER}})) &\stackrel{\mathrm{def}}{=} \mathit{is\text{-}intr\text{-}timer}(c_{\mathrm{TIMER}}), \\ &\dots \\ \mathit{is\text{-}intr\text{-}dev}(\mathit{idle\text{-}dev}) &\stackrel{\mathrm{def}}{=} \mathsf{F}. \end{aligned}$$

3.3.4 Devices Systems

Several devices can be organized in a devices system. Without loss of generality, in this thesis we consider models with 8 devices at most. Let $\mathit{Devnum} = \{1, \dots, 8\}$ be the set of possible device identifiers. For all device address *addr* we use notation $\langle \mathit{addr} \rangle_{\mathit{Devnum}}$ to extract device identifiers. The definition depends on a particular coupling of devices with the processor. See the next section for details.

Devices systems are modeled as mappings from device identifiers to generalized device configurations:

$$c_{\mathrm{DS}} :: C_{\mathrm{DS}} \stackrel{\mathrm{def}}{=} \mathit{Devnum} \mapsto C_{\mathrm{GD}}.$$

We distinguish two possible kinds of transitions a device may take in a system: internal, which are taken as a reaction to memory-interface input from the processor, and external, which process the inputs from the external environment.

In an internal step it is defined by the memory-interface port address which of the devices in the system makes a transition. As we consider two representations, natural and bit vector, of the memory interface, next, we define two functions for internal steps of a device in the system.

The internal step of a device in a system with the bit-vector representation of memory interface inputs

$$\delta_{\mathrm{DS}}^{\mathrm{INT}.Bv} :: C_{\mathrm{DS}} \times \mathit{Mifi}_{Bv} \mapsto C_{\mathrm{DS}} \times \mathit{Mifo}_{Bv}$$

is defined as follows. Let c_{DS} be a devices system, let *mifi* be an input from memory interface, and let $did = \langle mifi.a \rangle_{Devnum}$ be a device number specified by access address *mifi.a*. The device *did* performs a step:

$$\delta_{\mathrm{GD}}(c_{\mathrm{DS}}(did), \mathit{mifi\text{-}bv\text{-}to\text{-}nat}(mifi), \varepsilon\text{-}\mathit{eifi}(c_{\mathrm{DS}}(did))) \;=\; (c_{\mathrm{GD}}, mifo),$$

and the resulting devices system is:

$$\delta_{\mathrm{DS}}^{\mathrm{INT}.Bv}(c_{\mathrm{DS}}, mifi) \stackrel{\mathrm{def}}{=} (c'_{\mathrm{DS}}, bin(mifo))$$

with

$$c'_{\mathrm{DS}}(i) \;=\; \begin{cases} c_{\mathrm{GD}} & \text{if } i = did \\ c_{\mathrm{DS}}(i) & \text{otherwise} \end{cases}.$$

Completely analogously we define the internal step of a device in a system with respect to the natural memory interface:

$$\delta_{\mathrm{DS}}^{\mathrm{INT}.\mathbb{N}} :: C_{\mathrm{DS}} \times \mathit{Mifi}_{\mathbb{N}} \mapsto C_{\mathrm{DS}} \times \mathit{Mifo}_{\mathbb{N}}.$$

For the external step an explicit input of an identifier *did* of the device which is supposed to make a step is necessary:

$$\delta_{\mathrm{DS}}^{\mathrm{EXT}} :: C_{\mathrm{DS}} \times \mathit{Devnum} \times \mathit{Eifi}_{\mathrm{GD}} \mapsto C_{\mathrm{DS}},$$

$$\delta_{\mathrm{DS}}^{\mathrm{EXT}}(c_{\mathrm{DS}}, did, eifi) \stackrel{\mathrm{def}}{=} c'_{\mathrm{DS}},$$

where

$$c'_{\mathrm{DS}}(i) \;=\; \begin{cases} \delta_{\mathrm{GD}}(c_{\mathrm{DS}}(did), \varepsilon\text{-}\mathit{mifi}_{\mathbb{N}}, eifi).dev & \text{if } i = did \\ c_{\mathrm{DS}}(i) & \text{otherwise} \end{cases}.$$

Interrupts. For the whole devices system we define a function that computes the interrupt level for all devices in a generalized device configuration. The function

$$\mathit{intr\text{-}dev\text{-}bv} :: C_{\mathrm{DS}} \mapsto Bv$$

returns a bit vector in which the n-th bit indicates the interrupt level of the device with identifier n:

$$\mathit{intr\text{-}dev\text{-}bv}(c_{\mathrm{DS}}) \stackrel{\mathrm{def}}{=} [\mathit{bool2bit}(\mathit{is\text{-}intr\text{-}dev}(c_{\mathrm{DS}}(8))), \ldots, \mathit{bool2bit}(\mathit{is\text{-}intr\text{-}dev}(c_{\mathrm{DS}}(1)))].$$

Keeping in mind that we have 8 devices and the vector of external interrupts to the processor is of length 19, we extend the vector of devices interrupts with a vector of 11 zeros:

$$\mathit{intr\text{-}dev\text{-}bv}'(c_{\mathrm{DS}}) \stackrel{\mathrm{def}}{=} 0^{11} \circ \mathit{intr\text{-}dev\text{-}bv}(c_{\mathrm{DS}}).$$

3.4 Combined Systems

In this section we show how one can combine concurrent computational sources: processors and devices. We introduce a notion of a combined system, a processor model coupled with a devices system. Computations of such systems are guided by an external oracle, called an execution sequence, which defines for each point of time which of the computational sources, either the processor or some device, makes a step. Whenever a

device makes a step the external oracle additionally provides a device input. We define two particular kinds of combined systems: (i) VAMP ISA with devices, and (ii) VAMP assembly with devices. The reader should consults theses of Knapp [64] and Tverdyshev [115] for details on the first model, and of Alkassar [5] for the second. As we consider memory-mapped devices in both models, interaction of the processor with devices requires adjustment of load and store instructions semantics: we reserve a portion of processor's memory accessing which, we actually access devices.

3.4.1 Coupling Processors with Devices

Devices system introduced in the end of Section 3.3 can be coupled with a processor model. We refer to a resulting system as a *combined system*.

Execution Sequences and External Inputs

Independently of a processor model used, e.g., VAMP ISA or VAMP assembly, the reasoning about combined systems involves a notion of execution sequences. An execution sequence is an external oracle which parametrizes runs of a combined systems. At each step of the run the execution sequence defines whether the processor or a particular device performs a transition. As mentioned in Section 3.3, a device needs an input from the external environment in order to make a step. So, the execution sequence delivers an appropriate external input.

A sequence element denotes whether a processor or a device with a particular number and particular input makes a step. Formally, a sequence element s is an instance of the inductive data type *SeqEl*:

$$\begin{array}{rcl} \mathit{SeqEl} & \stackrel{\text{def}}{=} & \mathsf{Proc} \\ & | & \mathsf{Dev}(\mathbb{N} \times \mathit{Eifi}_{\text{DEV}}). \end{array}$$

Then, a sequence *seq* is defined as a mapping from combined system's step numbers to sequence elements:

$$\mathit{Seq} \stackrel{\text{def}}{=} \mathbb{N} \mapsto \mathit{SeqEl}.$$

Not all execution sequence generated by an external oracle are suitable for our needs. Sequences that we are interested in must guarantee liveness of a processor and all devices. The processor or device liveness means that for any position in an execution sequence the processor or device eventually makes a step. Next, at any position where a device makes a step, an external input exactly for this device type must be delivered. Formally, an execution sequence *seq* is called valid, if (i) it is live with respect to the processor and all devices, i.e., for any position in the sequence there are positions further in the sequence at which the processor and devices make a step, and (ii) the devices inputs are well-typed, i.e., for any position n in the sequence where a device makes a step, an external input which matches that device type is delivered:

$$\begin{array}{rcl} \mathit{seq}\surd(\mathit{seq}, c_{\text{DS}}) & \stackrel{\text{def}}{=} & \forall\, n :\ \exists\, i :\ n < i \ \wedge\ \mathit{seq}(i) = \mathsf{Proc} \\ & \wedge & \forall\, dn, n :\ \exists\, i, \mathit{eifi}_{\text{GD}} :\ n < i \ \wedge\ \mathit{seq}(i) = \mathsf{Dev}(dn, \mathit{eifi}_{\text{GD}}) \\ & \wedge & \forall\, n :\ \mathit{seq}(n) = \mathsf{Dev}(dn, \mathit{eifi}_{\text{GD}}) \\ & & \qquad \longrightarrow\ \mathit{is\text{-}eifi\text{-}match\text{-}dev}(c_{\text{DS}}(dn), \mathit{eifi}_{\text{GD}}). \end{array}$$

Next, we introduce two functions in order to separate computations of the processor and the devices system.

Definition 3.17 (Processor step numbers) Let *seq* be an execution sequence and let T be a number of steps of a combined system. The function

$$\textit{proc-steps} :: \textit{Seq} \times \mathbb{N} \mapsto \mathbb{N}$$

examines an execution sequence prefix of length T and returns the number of processor steps in it:

$$\textit{proc-steps}(\textit{seq}, T) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } T = 0 \\ \textit{proc-steps}(\textit{seq}, T-1) + 1 & \text{if } \textit{seq}(T-1) = \mathsf{Proc} \\ \textit{proc-steps}(\textit{seq}, T-1) & \text{otherwise} \end{cases} .$$

Isabelle: `VAMPisaDevices/dlxifspec_dev_hd.proc_step_number`

Definition 3.18 (Devices inputs filter) Let *seq* be an execution sequence, let *did* be a device number, and let T be a number of steps of a combined system. The function

$$\textit{dev-input} :: \textit{Seq} \times \textit{Devnum} \times \mathbb{N} \mapsto \textit{Eifi}_{\text{DEV}}{}^{*}$$

examines an execution sequence prefix of length T and returns a list of inputs corresponding to the device with number *did*:

$$\textit{dev-input}(\textit{seq}, \textit{did}, T) \stackrel{\text{def}}{=} \begin{cases} [] & \text{if } T = 0 \\ \textit{dev-input}(\textit{seq}, \textit{did}, T-1) \circ [\textit{eifi}_{\text{GD}}] & \text{if } \textit{seq}(T-1) = \mathsf{Dev}(\textit{did}, \textit{eifi}_{\text{GD}}) \\ \textit{dev-input}(\textit{seq}, \textit{did}, T-1) & \text{otherwise} \end{cases} .$$

Isabelle: `VAMPisaDevices/dlxifspec_dev_hd.dev_step_times`
Note that, the length of this list gives us the number of steps for particular device.

Non-Interference of Devices

As defined in the end of Section 3.3 devices may take transitions triggered by the processor they interact with or by the external environment. As long as devices take steps triggered only by the external environment it is possible to split a computation of the combined system into two independent execution sequences of the processor and of the devices system. This allows us to reason about the processor computation separately and then exploit the result of such reasoning in order to describe the computation of the whole combined system.

In order to formalize this idea we define a predicate that compares two configurations of devices systems c_{DS} and c'_{DS} and determines whether c'_{DS} is obtained from c_{DS} by taking only external steps. The execution is guided by an execution sequence which contains also processor steps.

Definition 3.19 (Non-interference of devices) Let c_{DS} and c'_{DS} be configurations of a devices system, let *seq* be an execution sequence, and let T be a number of steps. The predicate

$$\textit{non-interf-dev} :: C_{\text{DS}} \times C_{\text{DS}} \times \textit{Seq} \times \mathbb{N} \mapsto \mathbb{B}$$

checks whether the configuration of the devices system c'_{DS} is obtained from the configuration c_{DS} by executing independently all devices steps contained in the prefix of the sequence *seq* of length T:

$$\textit{non-interf-dev}(c_{\mathrm{DS}}, c'_{\mathrm{DS}}, \textit{seq}, T) \stackrel{\mathrm{def}}{=} \forall\, \textit{did}:\ c'_{\mathrm{DS}}(\textit{did}) = \delta^*_{\mathrm{GD}}(c_{\mathrm{DS}}(\textit{did}), \varepsilon\text{-}\textit{mif}_{\mathbb{N}}^{|\textit{dev-input}(\textit{seq},\textit{did},T)|}, \textit{dev-input}(\textit{seq}, \textit{did}, T)).\textit{dev}$$

Isabelle: `VAMPisaDevices/dlxifspec_dev_hd`

Memory Mapping for Devices.

To access devices we map devices ports to some memory region. We define the constant

$$\texttt{DEVICES\_BORDER} \stackrel{\mathrm{def}}{=} \langle 1^{17}0^{15}\rangle$$

which partitions the VAMP assembly memory into two parts: normal memory with the addresses below this constant and devices region with the addresses above this constant. The next two predicates help us to find out to which part an address a belongs:

$$\textit{is-mem-addr}(a) \stackrel{\mathrm{def}}{=} a < \texttt{DEVICES\_BORDER},$$

$$\textit{is-dev-addr}(a) \stackrel{\mathrm{def}}{=} a \geq \texttt{DEVICES\_BORDER}.$$

Semantics of combined systems distinguishes whether the processor accesses some device by executing a load or store instruction with an effective address which belongs to devices ports. Let a be a device address represented by a bit vector, i.e., $a[31:15] = 1^{17}$. The corresponding device identifier is encoded by the bits from 14 to 12

$$\langle a\rangle_{\mathit{Devnum}} \stackrel{\mathrm{def}}{=} \langle a[14:12]\rangle.$$

The following bits (from 11 to 2) denote the port number $\langle a[11:2]\rangle$. To couple a processor with a devices system we also need to slightly adjust the semantics of memory access instructions.

3.4.2 VAMP ISA with Devices

Combined systems which use the VAMP ISA model as the processor are referred to as *VAMP ISA with devices* or *VAMP ISA combined systems*.

Configurations

Configurations $c_{\mathrm{ISA+DS}}$ of VAMP ISA with devices are represented with the record $C_{\mathrm{ISA+DS}}$ which has two fields:

- the processor $c_{\mathrm{ISA+DS}}.\textit{cpu} :: C_{\mathrm{ISA}}$, and
- the devices system $c_{\mathrm{ISA+DS}}.\textit{devs} :: C_{\mathrm{DS}}$.

Semantics

First of all, we need to adapt the interrupts definitions. Since we have two memory parts, it has to be guaranteed that page tables and the fetched instruction do not lie behind `DEVICES_BORDER`. We extend the instruction page fault predicate with a condition that an interrupt occurs if the fetch address or, in case of user mode, the corresponding page table entry address, belongs to the devices range. For load/store we allow the accessed address to point to a device port only if the memory operation has the width of a word.

To distinguish between devices access and normal instruction computations we introduce the predicate $\textit{is-dev-acc}_{\mathrm{ISA}}(c_{\mathrm{ISA}})$ which holds in case a load word or a store word instruction takes place with the effective address corresponding to a device port, denoted by $\textit{is-dev-addr}(\langle \textit{ea}(c_{\mathrm{ISA}})\rangle)$. In case a device access takes place appropriate memory interface inputs have to be generated by the processor. This is done by means of the function

$$\textit{make-mifi}_{\mathrm{ISA}} :: C_{\mathrm{ISA}} \mapsto \textit{Mifi}_{Bv},$$

which distinguishes cases for read and write instructions. The reader can consult `VAMPisa/dlxifspec.make_mifi` for the definition in Isabelle.

Semantics of load and store instructions is slightly adjusted in case a device access takes place. Load instructions save the memory interface output in the destination general purpose register. Store instructions simply increase program counters in this case — data which has to be written to a devices is stored in the memory interface input. Thus, we extend the signature of the VAMP ISA transition function with a memory interface output:

$$\delta_{\mathrm{ISA}} :: C_{\mathrm{ISA}} \times Bv \times \textit{Mifo}_{Bv} \mapsto C_{\mathrm{ISA}}.$$

Transition Function

The transition function of the VAMP ISA with devices model

$$\delta_{\mathrm{ISA+DS}} :: \mathbb{N} \times C_{\mathrm{ISA+DS}} \times \textit{Seq} \mapsto C_{\mathrm{ISA+DS}}$$

takes as arguments a number of steps T to be executed, an ISA combined system configuration $c_{\mathrm{ISA+DS}}$, and an execution sequence *seq* together with external inputs. The step function is defined by induction on the step numbers T. For $T = 0$ we have:

$$\delta^0_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \textit{seq}) \stackrel{\mathrm{def}}{=} c_{\mathrm{ISA+DS}}.$$

In the definition for the step $T + 1$, first we perform T steps of system:

$$(c^T_{\mathrm{ISA}}, c^T_{\mathrm{DS}}) \;=\; \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \textit{seq}),$$

then, depending on the current sequence element $s = \textit{seq}(T)$ the definition for the step $T + 1$ distinguishes three cases:

- the processor makes a device access: the new states c^{T+1}_{ISA} and c^{T+1}_{DS} of both the processor and the devices system (internal step) are computed,
- the processor makes a step without a device access: only the processor new state c^{T+1}_{ISA} is generated, and
- the devices system makes a step triggered by the external environment (external step): only the new state c^{T+1}_{DS} of the devices system is generated.

Formally:

$$\delta^{T+1}_{\text{ISA+DS}}(c_{\text{ISA+DS}}, seq) \stackrel{\text{def}}{=}$$

$$\begin{cases} (c^{T+1}_{\text{ISA}}, c^{T+1}_{\text{DS}}) & \text{if} \quad s = \mathit{Proc} \\ & \quad \wedge \ \mathit{is\text{-}dev\text{-}acc}_{\text{ISA}}(c^{T}_{\text{ISA}}) \\ & \quad \wedge \ (c^{T+1}_{\text{DS}}, \mathit{mifo}) = \delta^{\text{INT}.Bv}_{\text{DS}}(c^{T}_{\text{DS}}, \mathit{make\text{-}mifi}_{\text{ISA}}(c^{T}_{\text{ISA}})) \\ & \quad \wedge \ c^{T+1}_{\text{ISA}} = \delta_{\text{ISA}}(c^{T}_{\text{ISA}}, \mathit{intr\text{-}dev\text{-}bv}(c^{T}_{\text{DS}}), \mathit{mifo}) \\ (c^{T+1}_{\text{ISA}}, c^{T}_{\text{DS}}) & \text{if} \quad s = \mathit{Proc} \\ & \quad \wedge \ \neg \mathit{is\text{-}dev\text{-}acc}_{\text{ISA}}(c^{T}_{\text{ISA}}) \\ & \quad \wedge \ c^{T+1}_{\text{ISA}} = \delta_{\text{ISA}}(c^{T}_{\text{ISA}}, \mathit{intr\text{-}dev\text{-}bv}(c^{T}_{\text{DS}}), \varepsilon\text{-}\mathit{mifo}_{Bv}) \\ (c^{T}_{\text{ISA}}, c^{T+1}_{\text{DS}}) & \text{if} \quad s = \mathit{Dev}(\mathit{did}, \mathit{eifi}) \\ & \quad \wedge \ c^{T+1}_{\text{DS}} = \delta^{\text{EXT}}_{\text{DS}}(c^{T}_{\text{DS}}, \mathit{did}, \mathit{eifi}) \end{cases} .$$

3.4.3 VAMP Assembly with Devices

We call combined systems that have the processor component instantiated with the VAMP assembly model *VAMP assembly with devices* or *VAMP assembly combined systems*.

Configurations

Configurations $c_{\text{ASM+DS}}$ of VAMP assembly with devices are represented with records $C_{\text{ASM+DS}}$. The record has two fields:

- the processor $c_{\text{ASM+DS}}.\mathit{cpu} :: C_{\text{ASM}}$, and
- the devices system $c_{\text{ASM+DS}}.\mathit{devs} :: C_{\text{DS}}$.

Semantics

Semantics of the VAMP assembly with devices model distinguishes whether a processor access devices or not in the same fashion as VAMP ISA combined systems do. The predicate $\mathit{is\text{-}dev\text{-}acc}_{\text{ASM}}(c_{\text{ASM}})$ evaluates to true in case the address of memory access belongs to the devices range. In order to handle processor-devices interaction a memory interface input to devices is generated by means of the function

$$\mathit{make\text{-}mifi}_{\text{ASM}} :: C_{\text{ASM}} \mapsto \mathit{Mifi}_{\mathbb{N}}$$

formally defined in Isabelle under `VAMPasmDevices/VAMPasmDevices.make_mifi`.

Modifications in the semantics of load and store instructions are done in the same fashion as for VAMP ISA. However, we do not adapt the existing VAMP assembly model, but create another one with the transition function:

$$\delta_{\text{ASM-dev}} :: C_{\text{ASM}} \times \mathit{Mifo}_{\mathbb{N}} \mapsto C_{\text{ASM}}.$$

Transition Function

The transition function for several steps of the model VAMP assembly with devices

$$\delta_{\text{ASM+DS}} :: \mathbb{N} \times C_{\text{ASM+DS}} \times \mathit{Seq} \mapsto C_{\text{ASM+DS}}$$

performs n steps guided by an execution sequence seq starting from a given configuration $c_{\text{ASM+DS}}$ and yields an updated configuration of the VAMP assembly with devices. It is defined inductively over the step number. For $n = 0$ we have:

$$\delta^0_{\text{ASM+DS}}(c_{\text{ASM+DS}}, seq) \stackrel{\text{def}}{=} c_{\text{ASM+DS}}.$$

In the definition for the step $n + 1$, first we perform n steps of the system:

$$(c^n_{\text{ASM}}, c^n_{\text{DS}}) = \delta^n_{\text{ASM+DS}}(c_{\text{ASM+DS}}, seq).$$

Then, depending on $s = seq(n + 1)$ as well as whether a device access, denoted by $\textit{is-dev-acc}_{\text{ASM}}(c_{\text{ASM+DS}}.cpu)$, takes place the function distinguishes the three following cases:

- the processor attempts a device access, and hence new configurations c^{n+1}_{ASM} and c^{n+1}_{DS} of the processor and the devices systems, respectively, are computed,
- the process makes a step which does not access any device: only the new configuration c^{n+1}_{ASM} of the processor is computed, and
- some device performs a transition, therefore only the updated configuration c^{n+1}_{DS} is obtained.

Formally:

$$\delta^{n+1}_{\text{ASM+DS}}(c_{\text{ASM+DS}}, seq) \stackrel{\text{def}}{=}$$

$$\begin{cases} (c^{n+1}_{\text{ASM}}, c^{n+1}_{\text{DS}}) & \text{if} \quad s = \mathit{Proc} \\ & \quad \wedge \ \textit{is-dev-acc}_{\text{ASM}}(c^n_{\text{ASM}}) \\ & \quad \wedge \ (c^{n+1}_{\text{DS}}, \textit{mifo}) = \delta^{\text{INT.}\mathbb{N}}_{\text{DS}}(c_{\text{DS}}, \textit{make-mifi}_{\text{ASM}}(c^n_{\text{ASM}})) \\ & \quad \wedge \ c^{n+1}_{\text{ASM}} = \delta_{\text{ASM-dev}}(c_{\text{ASM}}, \textit{mifo}) \\ (c^{n+1}_{\text{ASM}}, c^n_{\text{DS}}) & \text{if} \quad s = \mathit{Proc} \\ & \quad \wedge \ \neg \textit{is-dev-acc}_{\text{ASM}}(c^n_{\text{ASM}}) \\ & \quad \wedge \ c^{n+1}_{\text{ASM}} = \delta_{\text{ASM-dev}}(c_{\text{ASM}}, \varepsilon\textit{-mifo}_{\mathbb{N}}) \\ (c^n_{\text{ASM}}, c^{n+1}_{\text{DS}}) & \text{if} \quad s = \mathit{Dev}(\textit{did}, \textit{eifi}) \\ & \quad \wedge \ c^{n+1}_{\text{DS}} = \delta^{\text{EXT}}_{\text{DS}}(c^n_{\text{DS}}, \textit{did}, \textit{eifi}) \end{cases}.$$

3.5 C0 Programming Language

In this section we present an overview of C0, a type-safe garbage-collected dialect of C without pointer arithmetic. C0 was designed within Verisoft as a compromise between two orthogonal issues: the language has to be powerful enough to allow implementation of systems software while having clean formal semantics making verification of this software feasible. As systems software, and microkernels in particular, may access hardware resources beyond visibility of the C0 language, e.g., processor's registers, we extend C0 with an inline assembly statement referring to the resulting language as C0_{A}. We start the section by introducing main features of C0 as well as its limitations compared to standard C. We present formalization of C0 programs and show how their computations are model by the C0 small-step semantics. The most complete reference to the C0 language and its small-step semantics is the thesis of Leinenbach [66]. In this section we introduce a relatively detailed formalism for C0 programs and configurations because it is necessary for correctness theorems of CVM. The semantics is, however, described without many peculiarities.

3.5.1 Overview

The C0 programming language is a restricted version of ANSI C [1]. The C0 concrete syntax and visibility rules for variables are very similar to standard C. Operational semantics, though, is similar to Pascal. The major restrictions are:

- no initialization during declarations, except for a constant declaration,
- no side-effects inside expressions,
- no function calls inside expressions,
- the size of arrays is fixed at the compile time,
- no variable declarations in functions after the first statement,
- only one return statement which is the last control command in each function,
- no pointer arithmetic,
- no pointers to local variables,
- no pointers to functions,
- no void pointers, i.e. all pointers are typed.

The built-in type system of C0 is limited to four basic types, namely:

- 32-bit signed integers: `int` $= \{-2^{31}, \ldots, 2^{31} - 1\}$,
- 32-bit unsigned integers: `unsigned int` $= \{0, \ldots, 2^{32} - 1\}$,
- booleans: `bool` $= \{$`true`, `false`$\}$, and
- 8-bit signed integers: `char` $= \{-128, \ldots, 127\}$.

Based on these simple types one can construct complex types:

- typed pointers: `ty *x;`,
- arrays: `ty a[size];` and
- structures: `struct ty_1 ty_2 data;`.

The statements allowed in C0 language are:

- Assignment: `l = expr;`
  Besides elementary types and structures C0 allows assignment of array values, which is not possible in standard C.
- Loop: `while (cond) { stmts }`
- Conditional: `if (cond) { stmts_1 } else { stmts_2 }`
- Function call: `l = foo(expr_1, ..., expr_n);`
  Recursion is supported.
- Return: `return expr;`

- Allocation of dynamic memory: `l = new(typ);`
  There is no statement for deallocation of memory. Instead a garbage collector is supposed to used. However, in a microkernel we do not use it because memory deallocation is not needed.

- Inline assembly statement: `asm (instr_1, ..., instr_n);`

3.5.2 C0 Programs

A C0 program is identified by three entities:

- *the type environment*, a table which stores information about types used in the program,

- *the function table* which stores information about functions of the program, namely their parameters, types of return values, and statements constituting their bodies, and

- *the global symbol table* which stores names of program's global variables together with their types.

We formalize these concepts in the following order. We start with type environments. As function bodies constitute C0 statements which themselves use C0 expressions we need to formalize these two notions first in order to define a formal notion of a function. We define symbol tables which are lists of variables, like function parameter lists or global variables list. Having this we define function tables and conclude by defining the overall type for C0 programs.

Type Environments

C0 types are formalized by the following inductive data type

$$\begin{aligned} Ty \stackrel{\text{def}}{=} & \ \mathit{bool}_{\mathrm{T}} \\ | & \ \mathit{int}_{\mathrm{T}} \\ | & \ \mathit{char}_{\mathrm{T}} \\ | & \ \mathit{unsgnd}_{\mathrm{T}} \\ | & \ \mathit{ptr}_{\mathrm{T}}(\mathbb{S}) \\ | & \ \mathit{null}_{\mathrm{T}} \\ | & \ \mathit{arr}_{\mathrm{T}}(\mathbb{N} \times Ty) \\ | & \ \mathit{str}_{\mathrm{T}}((\mathbb{S} \times Ty)^*). \end{aligned}$$

A structural type is constructed from a list of pairs representing names of its components cn_i together with their types ty_i: $\mathit{str}_{\mathrm{T}}([(cn_0, ty_0), \ldots, (cn_n, ty_n)])$. For each structure element we will use the notation $.sfn$ to access the field's name and $.ty$ for its type. An array type is defined by the array size n and types of its elements ty': $\mathit{arr}_{\mathrm{T}}(n, ty')$. The parameter of a pointer type is the name of type tn it points to: $\mathit{ptr}_{\mathrm{T}}(tn)$.

We call a type ty elementary, denoted by $\mathit{is\text{-}elem}_{\mathrm{t}}(ty)$, if ty is not a structural or an array type. The size of a type is computed by the following function([66, Definition 4.1]):

$$\mathit{size}_{\mathrm{t}} :: Ty \mapsto \mathbb{N},$$

$$size_{\mathrm{t}}(ty) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \mathit{is\text{-}elem}_{\mathrm{t}}(ty) \\ n \cdot size_{\mathrm{t}}(ty') & \text{if } ty = \mathsf{arr}_{\mathrm{T}}(n, ty') \\ \sum_{j=0}^{|sn|-1} size_{\mathrm{t}}(sn[j].ty) & \text{if } ty = \mathsf{str}_{\mathrm{T}}(sn) \end{cases}.$$

A type environment (also called a type table) te is a list of pairs (tn, td) consisting of type names and types. Formally, type environments are instances of the type

$$Tenv \stackrel{\text{def}}{=} (\mathbb{S} \times Ty)^*.$$

Expressions

The most basic unit of C0 expressions are literals, i.e., symbols like "`1`", "`A`", or "`true`". Literals are modeled by the inductive data type *Lit* which has one constructor for each of the four C0 basic types and an additional constructor for null pointer literals. All constructors for the basic types have a single parameter of the corresponding type which denotes their value. Formally:

$$Lit \stackrel{\text{def}}{=} \mathsf{bool}(\mathbb{B}) \mid \mathsf{int}(\mathbb{Z}) \mid \mathsf{unsgnd}(\mathbb{N}) \mid \mathsf{char}(\mathbb{Z}) \mid \mathsf{null}.$$

The abstract data type *Expr* formalizes C0 expressions:

$$\begin{aligned} Expr \stackrel{\text{def}}{=}\ & \mathsf{lit}(Lit) \\ \mid\ & \mathsf{var}(\mathbb{S}) \\ \mid\ & \mathsf{arr}(Expr \times Expr) \\ \mid\ & \mathsf{str}(Expr \times \mathbb{S}) \\ \mid\ & \mathsf{bin\text{-}op}(BinOp \times Expr \times Expr) \\ \mid\ & \mathsf{lazy\text{-}bin\text{-}op}(LazyOp \times Expr \times Expr) \\ \mid\ & \mathsf{un\text{-}op}(UnOp \times Expr) \\ \mid\ & \mathsf{addr\text{-}of}(Expr) \\ \mid\ & \mathsf{deref}(Expr). \end{aligned}$$

The first two cases construct a literal expression $\mathsf{lit}(l)$ and a variable access expression $\mathsf{var}(vn)$ from a literal l and a variable name vn, respectively. The constructor $\mathsf{arr}(e, i)$ is used to access an array expression e at an index i which is an expression itself. A structural expression e is accessed at the component with a name vn by means of the expression $\mathsf{str}(e, vn)$. Two expressions e_1 and e_2 connected together with a binary operator $bop :: BinOp$ or a lazy binary operator $lop :: LazyOp$ form expressions $\mathsf{bin\text{-}op}(bop, e_1, e_2)$ and $\mathsf{lazy\text{-}bin\text{-}op}(lop, e_1, e_2)$, respectively. The expression for a unary operator $uop :: UnOp$ applied to an expression e is $\mathsf{un\text{-}op}(uop, e)$. Binary, lazy binary, and unary operators supported by C0 are listed in tables 4.2–4.4 of Leinenbach's thesis [66]. The last two constructors are used for the address-of operation and dereference.

Statements

Statements in the C0 small-step semantics are annotated with unique statement identifiers represented by natural numbers. This information is used to determine the place of each statement in C0 programs. Statement identifiers will be denoted by *sid*, possibly with subscript indices.

We model statements with the following abstract data type:

$$
\begin{array}{rcl}
\mathit{Stmt} & \stackrel{\text{def}}{=} & \mathit{skip} \\
& | & \mathit{comp}(\mathit{Stmt} \times \mathit{Stmt}) \\
& | & \mathit{ass}(\mathit{Expr} \times \mathit{Expr} \times \mathbb{N}) \\
& | & \mathit{pAlloc}(\mathit{Expr} \times \mathbb{S} \times \mathbb{N}) \\
& | & \mathit{ifte}(\mathit{Expr} \times \mathit{Stmt} \times \mathit{Stmt} \times \mathbb{N}) \\
& | & \mathit{loop}(\mathit{Expr} \times \mathit{Stmt} \times \mathbb{N}) \\
& | & \mathit{sCall}(\mathbb{S} \times \mathbb{S} \times \mathit{Expr}^* \times \mathbb{N}) \\
& | & \mathit{esCall}(\mathbb{S} \times \mathbb{S} \times \mathit{Expr}^* \times \mathbb{N}) \\
& | & \mathit{xCall}(\mathbb{S} \times \mathit{Expr}^* \times \mathit{Expr}^* \times \mathbb{N}) \\
& | & \mathit{return}(\mathit{Expr} \times \mathbb{N}).
\end{array}
$$

$\mathrm{C0_A}$ programs additionally have an assembly statement with the constructor:

$$\mathit{asm}(\mathit{Instr}^* \times \mathbb{N}).$$

The empty statement is denoted *skip* and the sequential composition of two statements s and s' is $\mathit{comp}(s, s')$. These two statements are called structural and are not tagged with statement identifiers. For a compositional statement we define functions

$$\mathit{left\text{-}stmt}(\mathit{comp}(s_1, s_2)) \stackrel{\text{def}}{=} s_1 \qquad \mathit{right\text{-}stmt}(\mathit{comp}(s_1, s_2)) \stackrel{\text{def}}{=} s_2$$

to get the left and the right parts of the statement, respectively.

The assignment statement $\mathit{ass}(e, e', \mathit{sid})$ assigns the expression e the value of the expression e'. Dynamic memory allocation for a type ty is done by the statement $\mathit{pAlloc}(\mathit{ty}, e, \mathit{sid})$ which assigns the expression e the pointer to the newly allocated memory.

The conditional statement $\mathit{ifte}(e, s, s', \mathit{sid})$ executes either the statement s or s' depending on the value of the expression e. *While* loops are modeled by the statement $\mathit{loop}(e, s, \mathit{sid})$ which executes the loop body s while the boolean expression e holds.

Functions calls come in three flavors in the C0 small-step semantics. A normal call to a function with a name fn and parameters e_1 to e_n are represented by the statement $\mathit{sCall}(\mathit{vn}, \mathit{fn}, [e_1, \ldots, e_n], \mathit{sid})$. On termination of the function its return value is stored in the variable of name vn. An external call to an only declared function fn is modeled by the statement $\mathit{esCall}(\mathit{vn}, \mathit{fn}, [e_1, \ldots, e_n], \mathit{sid})$. An extended call, or an x-call, to the function of name fn is represented by the statement $\mathit{xCall}(\mathit{fn}, [e_1, \ldots, e_n], [e'_1, \ldots, e'_m], \mathit{sid})$ where e_i are expressions denoting the parts of the C0 state that can be changed by the x-call, and e'_i are expressions that are used as its input parameters. For a normal call $s = \mathit{sCall}(\mathit{vn}, \mathit{fn}, \mathit{el}, \mathit{sid})$ and an external call $s = \mathit{esCall}(\mathit{vn}, \mathit{fn}, \mathit{el}, \mathit{sid})$ we define several functions

$$\mathit{called\text{-}func}(s) \stackrel{\text{def}}{=} \mathit{fn} \qquad \mathit{param\text{-}list}(s) \stackrel{\text{def}}{=} \mathit{el} \qquad \mathit{l\text{-}var}(s) \stackrel{\text{def}}{=} \mathit{vn},$$

which extract parts of the callee's signature: (i) the name of the called function, (ii) the list of callee's parameters, and (iii) the name of the left-hand side variable.

Return statements with a return expression e are modeled by $\mathit{return}(e, \mathit{sid})$.

Finally, assembly statements $\mathit{asm}(\mathit{il}, \mathit{sid})$ receive as a parameter a list il of VAMP assembly instructions which has to be executed.

For each statement constructor we define a predicate which tests whether a statement s is constructed by this particular constructor. For instance this definition for the case of the return statement is:

$$\mathit{is\text{-}}\mathsf{return}(s) \stackrel{\text{def}}{=} \exists e, sid : \ s = \mathsf{return}(e, sid).$$

Next, we define several functions over C0 statements. Observe that compositional statements define a binary tree of the statements they compose. Sometimes it is not convenient to work with such trees — therefore, we define a more suitable representation of statement composition. The function $s2l :: \mathit{Stmt} \mapsto \mathit{Stmt}^*$ takes a statement s as an argument and, in case s is a compositional statement $\mathsf{comp}(s_1, s_2)$, flattens it into a list of statements by means of a left-side tree traversal:

$$\mathit{s2l}(\mathsf{comp}(s_1, s_2)) \stackrel{\text{def}}{=} \mathit{s2l}(s_1) \circ \mathit{s2l}(s_2).$$

For all other statements we define the list construction $\mathit{s2l}(s) \stackrel{\text{def}}{=} [s]$. A version of the function that additionally ignores empty statements is $\mathit{s2l}_{\text{ns}} :: \mathit{Stmt} \mapsto \mathit{Stmt}^*$: the definition differs only in case of the skip statement:

$$\mathit{s2l}_{\text{ns}}(\mathsf{skip}) \stackrel{\text{def}}{=} [].$$

The number of top-level return statements in a list of statements is computed by the function $\#\mathit{ret}_{\text{top}} :: \mathit{Stmt}^* \mapsto \mathbb{N}$:

$$\#\mathit{ret}_{\text{top}}(\mathit{sl}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \mathit{sl} = [] \\ \#\mathit{ret}_{\text{top}}(t) + 1 & \text{if } \mathit{sl} = [h] \circ t \wedge \mathit{is\text{-}}\mathsf{return}(h) \\ \#\mathit{ret}_{\text{top}}(t) & \text{otherwise} \end{cases}.$$

Note, that this function is not recursive for loop and conditional statements.

Function Tables

We call a list of variable names together with their types a *symbol table*. Symbol tables are modeled with the type

$$\mathit{Symtable} \stackrel{\text{def}}{=} (\mathbb{S} \times \mathit{Ty})^*.$$

For each element of a symbol table we will use the notation $.vn$ to access the variable name and $.ty$ for its type.

Information about a single C0 function is stored in a record of the type *Func*. Instances f of this type have four components:

- $f.\mathit{body} :: \mathit{Stmt}$: the body of the function,
- $f.\mathit{params} :: \mathit{Symtable}$: the list of function parameters,
- $f.\mathit{rtype} :: \mathit{Ty}$: the return type of the function, and
- $f.\mathit{lvars} :: \mathit{Symtable}$: the list of local variables of the function.

The symbol table for a particular function is constructed as a concatenation of its parameters and local variables:

$$\mathit{st}_{\text{fun}} :: \mathit{Func} \mapsto \mathit{Symtable},$$

$$st_{\text{fun}}(f) \stackrel{\text{def}}{=} f.params \circ f.lvars.$$

A function table is a list of pairs (fn, fd) constituting a function name and a function definition modeled by the record *Func*. Formally, we represent function tables with the type

$$Functable \stackrel{\text{def}}{=} (\mathbb{S} \times Func)^*.$$

Programs

C0 programs are defined by their type environments, function tables, and symbol tables of global variables. Thus, we represent C0 programs with the type Π_{C0}, whose elements π are records with three components:

- $\pi.te :: Tenv$: the type environment containing all new types defined in the program,
- $\pi.ft :: Functable$: the function table containing function names and definitions, and
- $\pi.gst :: Symtable$: the table of global variables.

3.5.3 Translatable C0 Programs

C0 programs are subject to compilation into VAMP assembly. The code generation algorithm designed in Verisoft does not work with arbitrary C0 programs. For instance if a program contains an expression which uses to many registers the code generation will not work. Therefore, a number of restrictions are imposed on expressions, statements, and programs such that:

- the generated code respects the necessary requirements to the size of immediate operands, and
- there are sufficiently many free VAMP assembly registers to evaluate the expressions.

C0 programs that fulfill these properties are called *translatable*.

Leinenbach formalizes this notion in Section 7.6 of his thesis [66] by defining the set $xltbl_{\text{prog}}$ of C0 programs which obey the above condition. We omit formalization here — for details the reader should examine Definition 7.41 of [66]. Formally we will denote that a program π is translatable by:

$$\pi \in xltbl_{\text{prog}}.$$

3.5.4 C0 Small-Step Semantics Configurations

Run-time information about execution of C0 programs is stored in C0 small-step semantics configurations. These configurations have two components:

- the memory configuration which stores information about C0 variables and their values, and
- the program rest which contains statements which still have to be executed.

C0 small-step semantics configurations are also called *C0 machines*.

Memory Configuration

C0 memory configuration has three parts for storing global, heap, and local variables together with return destinations. Each part is defined by the corresponding memory frames. Local variables correspond to a list of memory frames where each item defines variables together with their values of the function call stack. Before introducing memory frames and memory configurations formally we consider a notion of *generalized variables*.

Generalized variables. The pointers in the C0 small-step semantics are represented with the help of generalized variables or shortly g-variables. G-variables are modeled with the following inductive data type:

$$\begin{array}{rcl} \mathit{Gvar} & \stackrel{\text{def}}{=} & \mathit{gvar}_{\text{gm}}(\mathbb{S}) \\ & | & \mathit{gvar}_{\text{lm}}(\mathbb{N} \times \mathbb{S}) \\ & | & \mathit{gvar}_{\text{hm}}(\mathbb{N}) \\ & | & \mathit{gvar}_{\text{arr}}(\mathit{Gvar}, \mathbb{N}) \\ & | & \mathit{gvar}_{\text{str}}(\mathit{Gvar}, \mathbb{S}). \end{array}$$

Three base cases of g-variables represent a global variable with the name vn: $\mathit{gvar}_{\text{gm}}(vn)$, a local variable of name vn in the i-th local memory frame: $\mathit{gvar}_{\text{lm}}(i, vn)$, and a nameless heap variable identified by its number i: $\mathit{gvar}_{\text{hm}}(i)$. The inductive cases comprise constructors for array and structural g-variables. If g is a g-variable of an array type then the i-th array element $\mathit{gvar}_{\text{arr}}(g, i)$ is a g-variable as well. If g is a structural g-variable then a component $\mathit{gvar}_{\text{str}}(g, cn)$ with the name cn is also a g-variable.

Memory cells. An explicit, flat memory model which stores memory contents as a mapping from addresses represented by natural numbers to memory cells is used in the C0 small-step semantics. A single memory cell stores the values of a variable of an elementary type. Values of aggregate types are stored consecutively as sequences of memory cells. We model memory cells by the following data type with one constructor per elementary type:

$$\mathit{Mcell} \stackrel{\text{def}}{=} \mathit{int}(\mathbb{Z}) \mid \mathit{nat}(\mathbb{N}) \mid \mathit{char}(\mathbb{Z}) \mid \mathit{bool}(\mathbb{B}) \mid \mathit{ptr}(\mathit{Gvar}_{\perp}).$$

Pointers are represented by generalized variable type extended with a $\perp$-state denoting null pointers.

In order to get the value from a memory cell we use the following conversion functions:

$$\begin{array}{lcl} \mathit{m2b}(\mathit{bool}(b)) & \stackrel{\text{def}}{=} & b, \\ \mathit{m2u}(\mathit{nat}(n)) & \stackrel{\text{def}}{=} & n, \\ \mathit{m2i}(\mathit{int}(i)) & \stackrel{\text{def}}{=} & i, \\ \mathit{m2ch}(\mathit{char}(ch)) & \stackrel{\text{def}}{=} & ch, \\ \mathit{m2p}(\mathit{ptr}(p)) & \stackrel{\text{def}}{=} & p. \end{array}$$

Memory frames. *A memory frame* $m :: \mathit{Mframe}$ is a record with three components:

- the content of the memory frame: $m.ct :: \mathbb{N} \mapsto \mathit{Mcell}$,

- the symbol table $m.st :: \mathit{Symtable}$, a list of all variables of the memory frame together with their types, and
- the set of variables which are already initialized: $m.\mathit{init} :: 2^{\mathbb{S}}$.

Memory configuration. *A memory configuration* $mc :: \mathit{Memconf}$ is a record with three components:

- the global memory frame $mc.gm :: \mathit{Mframe}$,
- the list of memory frames and g-variables representing a stack of local memories $mc.lm :: (\mathit{Mframe} \times \mathit{Gvar})^*$ — each memory frame $mc.lm[i].mfr$ stores the values of local variables and each g-variable $mc.lm[i].res$ stores the return destination of a stack frame, and
- the heap memory frame $mc.hm :: \mathit{Mframe}$.

Note, that the symbol table of a heap memory frame has empty identifiers because heap variables are nameless. Moreover, the set of initialized variables is empty, but all heap variables are initialized during their allocation.

The local memory frames list grows as a new frame is inserted at its beginning. Hence, the i-th allocated frame is counted starting from the end. We introduce the following notation for local memory frame access:

$$lm[\![i]\!] \stackrel{\text{def}}{=} lm[|lm| - 1 - i].$$

We define the top local memory frame of a memory configuration mc and the top result variable as

$$lm_{\text{top}}(mc) \stackrel{\text{def}}{=} mc.lm[\![|mc.lm| - 1]\!].mfr = mc.lm[0].mfr,$$

$$res_{\text{top}}(mc) \stackrel{\text{def}}{=} mc.lm[\![|mc.lm| - 1]\!].res = mc.lm[0].res.$$

Top local, global, and heap symbol tables are extracted by means of the following functions:

$$\begin{aligned} lst_{\text{top}}(mc) &\stackrel{\text{def}}{=} lm_{\text{top}}(mc).st, \\ gst(mc) &\stackrel{\text{def}}{=} mc.gm.st, \\ hst(mc) &\stackrel{\text{def}}{=} mc.hm.st. \end{aligned}$$

Symbol configuration. Sometimes in this thesis we are not interested in the memory content and thus do not need the complete memory configuration. For some special needs it suffices to have only symbol tables of all memory frames. For this we introduce a concept of *symbol configurations* which are records $sc :: \mathit{Symconf}$ which are records with three components:

- $sc.gst :: \mathit{Symtable}$: the symbol table of the global memory frame,
- $sc.lst :: \mathit{Symtable}^*$: the list of symbol tables for the stack of local memories, and
- $sc.hst :: \mathit{Symtable}$: the symbol table of the heap memory frame.

Extraction of the symbol configuration from a memory configuration mc is done with the function $sc(mc)$.

Program Rest

The program rest remembers statements which still have to be executed. Initialized with the body of the main function it grows or shrinks depending on the execution of the program. Formally, the program rest is an instance of the type *Stmt*.

Configuration

Having formal definitions of the C0 memory configuration and the program rest we can define the C0 small-step configurations. Configurations c_{C0} are modeled with the record C_{C0} which has two components:

- the memory configuration $c_{\mathrm{C0}}.mem :: \mathit{Memconf}$, and
- the program rest $c_{\mathrm{C0}}.prog :: \mathit{Stmt}$.

Occasionally, we will need a version of C0 configuration which additionally encapsulates a respective C0 program. We extend the type for C0 configuration as follows:

$$c_{\mathrm{C0}} :: C_{\mathrm{C0}}^{\mathrm{mono}} \stackrel{\mathrm{def}}{=} (te, ft, mem, prog)$$

and call the result *monolithic C0 configurations*. Note that while a type table and a function table are explicitly inserted in the record, the global symbol table is already included in the global memory frame.

The current statement of a C0 configuration. For a C0 small-step semantics configuration c_{C0} let the following function extracts the statement that has to be currently executed:

$$\mathit{stmt}(c_{\mathrm{C0}}) \stackrel{\mathrm{def}}{=} \mathit{s2l}(c_{\mathrm{C0}}.prog)[0]$$

3.5.5 Initial Configuration

C0 initial configuration defines the default values of all its component. In this work we do not need a complete formal definition of C0 initial configuration. We restrict ourselves to definitions of initial global and heap memory frames.

The function $\mathit{init}_{\mathrm{val}} :: \mathit{Ty} \mapsto \mathit{Mcell}^*$ defines initial values for all C0 types by structural induction. Numeric types have zero as the initial value, the boolean type is initialized with the *false* constant F, and pointers with a null pointer. Array elements and structure components are initialized with default values of the corresponding types. For formulas consult Definition 4.28 of [66].

The function $\mathit{init}_{\mathrm{st}} :: \mathit{Symtable} \mapsto \mathit{Mcell}^*$ computes the initialized content for a given symbol table by concatenating default values of all its variables:

$$\mathit{init}_{\mathrm{st}}(st) \stackrel{\mathrm{def}}{=} \begin{cases} [] & \text{if } st = [] \\ \mathit{init}_{\mathrm{val}}(ty) \circ \mathit{init}_{\mathrm{st}}(st') & \text{if } st = (vn, ty) \circ st' \end{cases}.$$

The function $\mathit{init}_{\mathrm{ct}} :: \mathit{Symtable} \mapsto (\mathbb{N} \mapsto \mathit{Mcell})$ converts such list into memory content:

$$\mathit{init}_{\mathrm{ct}}(st)(i) \;=\; \mathit{init}_{\mathrm{st}}(st)[i]$$

Finally, the function $\mathit{init}_{\mathrm{vars}} :: \mathit{Mframe} \mapsto \mathit{Mframe}$ initializes all variables of a given memory frame. The obtained memory frame $m' = \mathit{init}_{\mathrm{vars}}(m)$ has the initialized content,

and all variables of the memory frame are added to the set of initialized variables:

$$\begin{aligned} m'.ct &= init_{\text{ct}}(m.st), \\ m'.init &= vns(m.st), \end{aligned}$$

where the function $vns :: Symtable \mapsto 2^{\mathbb{S}}$ collects names of the variables from the symbol table into a set:

$$vns(st) \stackrel{\text{def}}{=} \{x : \exists\, i :\ st[i].vn = x\}.$$

Initial memory frame for a symbol table st is computed with the function $init_{\text{mem}} :: Symtable \mapsto Mframe$, such that $init_{\text{mem}}(st) = m$ where

$$\begin{aligned} m.st &= st, \\ m.init &= \emptyset. \end{aligned}$$

The content component is undefined.

Having this, the initial value of a global memory frame is computed by the function

$$init_{\text{gm}}(gst) \stackrel{\text{def}}{=} init_{\text{vars}}(init_{\text{mem}}(gst)).$$

The initial value of a heap memory frame is a constant

$$init_{\text{hm}} \stackrel{\text{def}}{=} init_{\text{mem}}([]).$$

3.5.6 Valid C0 Small-Step Semantics Configurations

The definitions presented in this section so far allow us to encode even absurd programs containing statements like `5=true;`. In order to distinguish rational programs from ridiculous ones a validity predicate over C0 configurations has to be introduced. Leinenbach defines such a predicate formally in Chapter 5 of his thesis [66]. Since description of this formal definition consumes about 30 pages we will not copy any details here, but rather present a bird's-eye view of the predicate.

A C0 small-step semantics configuration c_{C0} is valid with respect to a type name environment te and a function table ft if the following holds:

1. the function table ft is valid with respect to the global symbol table, denoted by $ft \in valid_{\text{ft}}(te, gst(c_{\text{C0}}.mem))$,
2. the program rest of c_{C0} is valid,
3. the number of return statements in the program rest is less than the recursion depth of c_{C0},
4. the stack of c_{C0} is valid,
5. the type name environment of c_{C0} is valid, denoted by $te \in valid_{\text{tenv}}$,
6. the global symbol table of c_{C0} is valid, denoted by $gst(c_{\text{C0}}.mem) \in valid_{\text{st}}(te)$,
7. all local symbol tables of c_{C0} belong to some function in ft,
8. all types of all heap variables are valid types,
9. all memory frames of c_{C0} are type correct, and

10. the return destinations of c_{C0} are valid.

These criteria define the set $C0\surd(te, ft)$ of valid C0 small-step semantics configurations. Formally this set is introduced in Definition 5.38 of Leinenbach's thesis [66]. We denote that a configuration c_{C0} is valid by

$$c_{\mathrm{C0}} \in C0\surd(te, ft).$$

In spite of point 3, during verification we always need the fact that the number of return statements in the program rest is equal to the recursion depth minus one. So we extend the validity predicate $C0\surd$ with this condition, defining the predicate $C0'\surd$:

$$\begin{array}{ll} & c_{\mathrm{C0}} \in C0\surd(te, ft) \\ \wedge & \#ret_{\mathrm{top}}(s2l(c_{\mathrm{C0}}.prog)) + 1 = |c_{\mathrm{C0}}.mem.lm| \\ \longrightarrow & c_{\mathrm{C0}} \in C0'\surd(te, ft). \end{array}$$

Another important validity definition concerns g-variables. We say that a g-variable g is in the set of valid g-variables if it has a well-formed structure for a given symbol configuration sc([66, Definition 5.19]):

$$g \in gvars\surd(sc).$$

The set $reachable_{\mathrm{g}}(mc)$ contains all reachable valid g-variables of memory configuration mc:

$$g \in reachable_{\mathrm{g}}(mc).$$

3.5.7 Semantics

The transition function

$$\delta_{\mathrm{C0}} :: Tenv \times Functable \times C_{\mathrm{C0}} \mapsto C_{\mathrm{C0}\perp}$$

of the C0 small-step semantics maps, with respect to a type environment te and a function table ft, a configuration c_{C0} to its successor configuration c'_{C0}, such that $\lfloor c'_{\mathrm{C0}} \rfloor = \delta_{\mathrm{C0}}(te, ft, c_{\mathrm{C0}})$ or, in case of an error to $\perp$. The transition function is defined by induction on the program rest — thus, it distinguishes cases for each C0 statement.

Each case formalizes the statement's functionality described in this section before. We omit the formal definition of transition function — the reader should consult pages 61–67 of Leinenbach's thesis [66].

Leinenbach does not consider semantics of an external call statement *esCall*. External calls are introduced into C0 semantics in order model programs separated into several *modules*. These modules are C0 programs themselves where module M_1 invokes an external call $esCall(vn, fn, [e_1, \ldots, e_n], sid)$ to a function of name fn which is only declared in M_1 and is implemented in a module M_2. These two modules can be linked together on the source code level — see Section 6.2 for formal details on linking — in a single C0 programs which has no longer external calls. However, C0 small-step semantics does not forbid execution of modules, i.e., C0 programs with external call statements. The transition function δ_{C0} treats external call as a statement *skip*, however, the statement itself should be valid as a normal function call.

A multiple-step transition function

$$\delta_{\mathrm{C0}} :: \mathbb{N} \times Tenv \times Functable \times C_{\mathrm{C0}} \mapsto C_{\mathrm{C0}\perp}$$

written $\delta^n_{\mathrm{C0}}(te, ft, c_{\mathrm{C0}})$ is defined by induction on n:

$$\begin{array}{lcl} \delta^0_{\mathrm{C0}}(te, ft, c_{\mathrm{C0}}) & \stackrel{\mathrm{def}}{=} & \lfloor c_{\mathrm{C0}} \rfloor, \\ \delta^n_{\mathrm{C0}}(te, ft, c_{\mathrm{C0}}) & \stackrel{\mathrm{def}}{=} & \begin{cases} \delta_{\mathrm{C0}}(te\, ft, c'_{\mathrm{C0}}) & \text{if } \delta^{n-1}_{\mathrm{C0}}(te, ft, c_{\mathrm{C0}}) = \lfloor c'_{\mathrm{C0}} \rfloor \\ \bot & \text{if } \delta^{n-1}(te, ft, c_{\mathrm{C0}}) = \bot \end{cases}. \end{array}$$

For a monolithic configuration $c_{\mathrm{C0}} :: C^{\mathrm{mono}}_{\mathrm{C0}}$ we define a transition function with the help of a normal one:

$$\delta^{\mathrm{mono}}_{\mathrm{C0}} :: C^{\mathrm{mono}}_{\mathrm{C0}} \mapsto C^{\mathrm{mono}}_{\mathrm{C0}\bot},$$

$$\delta^{\mathrm{mono}}_{\mathrm{C0}}(c_{\mathrm{C0}}) \stackrel{\mathrm{def}}{=} \delta_{\mathrm{C0}}(c_{\mathrm{C0}}.te, c_{\mathrm{C0}}.ft, (c_{\mathrm{C0}}.mem, c_{\mathrm{C0}}.prog)).$$

3.5.8 Evaluation

Additionally to the defined semantics we also need a number of functions that give us such data as the type information and the addresses of variables, and the values of variables and expressions. Below we copy some definitions for that from Leinenbach's thesis[66].

Variables. We define ([66, Definition 4.14]) the *type* of a variable vn in a given symbol table st as:

$$type_{\mathrm{v}}(st, vn) \stackrel{\mathrm{def}}{=} \begin{cases} ty & \text{if } st = [(vn, ty)] \circ st' \\ type_{\mathrm{v}}(st', vn) & \text{if } st = [(vn', ty)] \circ st' \ \wedge \ vn' \neq vn \\ \mathsf{A} & \text{otherwise} \end{cases}.$$

We define ([66, Definition 4.13]) the *base address* of a variable vn in a given symbol table st as:

$$ba_{\mathrm{v}}(st, vn) \stackrel{\mathrm{def}}{=} \begin{cases} 0 & \text{if } st = [(vn, ty)] \circ st' \\ size_{\mathrm{t}}(ty) + ba_{\mathrm{v}}(st', vn) & \text{if } st = [(vn', ty)] \circ st' \ \wedge \ vn' \neq vn \\ \mathsf{A} & \text{otherwise} \end{cases}.$$

We define ([66, Definition 4.15]) the *type of a g-variable* as:

$$ty_{\mathrm{g}}(sc, g) \stackrel{\mathrm{def}}{=} \begin{cases} type_{\mathrm{v}}(sc.gst, vn) & \text{if } g = \mathsf{gvar}_{\mathrm{gm}}(vn) \\ type_{\mathrm{v}}(sc.lst[\![i]\!], vn) & \text{if } g = \mathsf{gvar}_{\mathrm{lm}}(i, vn) \\ sc.hst[i].ty & \text{if } g = \mathsf{gvar}_{\mathrm{hm}}(i) \\ ty & \text{if } g = \mathsf{gvar}_{\mathrm{arr}}(G, i) \ \wedge \ ty_{\mathrm{g}}(sc, G) = \mathsf{arr}_{\mathrm{T}}(ty, n) \\ type_{\mathrm{v}}(c, sn) & \text{if } g = \mathsf{gvar}_{\mathrm{str}}(G, sn) \ \wedge \ ty_{\mathrm{g}}(sc, G) = \mathsf{str}_{\mathrm{T}}(c) \end{cases}.$$

We define ([66, Definition 4.18]) the *base address of a g-variable* as:

$$ba_{\mathrm{g}}(sc, g) \stackrel{\mathrm{def}}{=} \begin{cases} ba_{\mathrm{v}}(sc.gst, vn) & \text{if } g = \mathsf{gvar}_{\mathrm{gm}}(vn) \\ ba_{\mathrm{v}}(sc.lst[\![i]\!], vn) & \text{if } g = \mathsf{gvar}_{\mathrm{lm}}(i, vn) \\ \sum_{j=0}^{i-1} size_{\mathrm{t}}(hst[j].ty) & \text{if } g = \mathsf{gvar}_{\mathrm{hm}}(i) \\ ba_{\mathrm{g}}(sc, G) + i \cdot size_{\mathrm{t}}(ty) & \text{if } g = \mathsf{gvar}_{\mathrm{arr}}(G, i) \ \wedge \ ty_{\mathrm{g}}(sc, G) = \mathsf{arr}_{\mathrm{T}}(ty, n) \\ ba_{\mathrm{g}}(sc, G) + ba_{\mathrm{v}}(c, sn) & \text{if } g = \mathsf{gvar}_{\mathrm{str}}(G, sn) \ \wedge \ ty_{\mathrm{g}}(sc, G) = \mathsf{str}_{\mathrm{T}}(c) \end{cases}.$$

For a memory configuration mc and an elementary g-variable g we define ([66, Definition 4.22]) its value as:

$$value_{\mathrm{g}}(mc, g) \stackrel{\mathrm{def}}{=} ct(ba_{\mathrm{g}}(sc(mc), g)),$$

where ct is $mc.gm.ct$ if g is a global variable, or $mc.lm[\![i]\!].mfr.ct$ if g is a local variable from the i-th frame, or $mc.hm.ct$ if g is a heap variable.

Expressions. The expression evaluation function $reval(te, mc, e)$ is defined inductively over an expression tree ([66, Section 4.3.4]). In case e is a variable the function $value_{\mathrm{g}}$ is used. Otherwise, equivalent abstract operations are applied.

The function $type(te, gst, lst_{\mathrm{top}}, e)$ computes the type of an expression.

The function *is-initialized*(te, mc, e) tests whether an expression is initialized; it boils down to the test that all variables used in the expression are initialized.

Chapter 4

Compiling C0 to VAMP

In the previous chapter we have defined three models for reasoning about programs: VAMP ISA, VAMP assembly, and C0 small-step semantics. Most of the software in Verisoft has been implemented and verified at the C0 level. However some key theorems of Verisoft, like the CVM correctness theorem, have to describe, among others, how a C0 program is executed on the target machine. In order to define this, C0 programs need to be translated — via assembly code — to the object code which is executable on the hardware. For this a simple non-optimizing compiler [66, 93] has been developed and verified in Verisoft.

The correctness theorem of the compiler specification [66] is a simulation theorem between a C0 program executed by the C0 small-step semantics and the generated assembly code executed by the VAMP assembly model. The goal of this chapter is to extend this theorem such that VAMP ISA becomes its target model.

Since only VAMP ISA has a mechanism to switch between the execution modes we can argue only about uninterrupted execution of a program in the system mode while discussing compilation of C0 to VAMP ISA. Correctness of user programs will be presented in the context of the running kernel (Chapter 10).

In Section 4.1 we introduce a theorem that the VAMP assembly model is simulated by the VAMP ISA model. Section 4.2 reviews the correctness theorem of the C0 compiler specification whose detailed description is presented in Leinenbach's thesis [66]. In Section 4.3 we combine these two statements into a single theorem: a C0 program executed by the C0 small-step semantics is simulated by the generated object code executed by the VAMP ISA model.

4.1 Simulation of VAMP Assembly by VAMP ISA

The simulation theorems between VAMP assembly and VAMP ISA come in two flavors — without and with devices access. The idea behind the theorems is as follows. We start with assembly and ISA configurations (with and without devices) between which an abstraction relation holds. We execute a given assembly program by the assembly model and find the corresponding number of steps of the ISA model (combined system), such that the abstraction relation is preserved.

In this section we define the abstraction relation mentioned above, state the necessary preconditions to the simulation, and present the respective simulation theorems.

4.1.1 Abstraction Relation

The major difference between the VAMP ISA and assembly models is data representation. ISA registers are defined as bit vectors while assembly registers are natural numbers. ISA memory is defined as a mapping from bit vectors of length 29 to pairs of bit vectors of length 32 while assembly memory is a mapping of 30-bit natural numbers to 32-bit integers. The abstraction relation between VAMP ISA and assembly basically defines how components of an ISA configuration are converted to their assembly equivalents.

Definition 4.1 (Assembly-ISA control equivalence) Let c_{ASM} and c_{ISA} be assembly and ISA machine configurations. The control equivalence *c-equiv* holds between the machines if values of respective program counters match:

$$\begin{aligned} \textit{c-equiv}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}) \quad &\stackrel{\mathrm{def}}{=} \quad c_{\mathrm{ASM}}.pc = \langle c_{\mathrm{ISA}}.pc \rangle \\ &\wedge \quad c_{\mathrm{ASM}}.dpc = \langle c_{\mathrm{ISA}}.dpc \rangle. \end{aligned}$$

Isabelle: `VAMPasm2isaSystem/equivalence.c_equiv`

Definition 4.2 (Assembly-ISA GPR equivalence) Let a and b be assembly and ISA general purpose register files. The files are equivalent, which is stated by the predicate

$$\textit{gpr-equiv} :: \mathit{Regf}_{\mathrm{ASM}} \times \mathit{Regf}_{\mathrm{ISA}} \mapsto \mathbb{B},$$

$$\textit{gpr-equiv}(a, b) \stackrel{\mathrm{def}}{=} \forall\, i \in Bv_5 :\ \textit{gpr-read}_{\mathrm{ASM}}(a, \langle i \rangle) = [\textit{gpr-read}_{\mathrm{ISA}}(b, i)]$$

if the values of their 32 items accessed by the VAMP ISA respectively assembly GPR read functions match.

Isabelle: `VAMPasm2isaSystem/equivalence.gprs_equiv`

Definition 4.3 (Assembly-ISA SPR equivalence) Let a and b be assembly and ISA special purpose register files. The files are equivalent, which is stated by the predicate

$$\textit{spr-equiv} :: \mathit{Regf}_{\mathrm{ASM}} \times \mathit{Regf}_{\mathrm{ISA}} \mapsto \mathbb{B},$$

$$\textit{spr-equiv}(a, b) \stackrel{\mathrm{def}}{=} \forall\, i \in Bv_5 :\ \textit{spr-read}_{\mathrm{ASM}}(a, \langle i \rangle) = [\textit{spr-read}_{\mathrm{ISA}}(b, i)]$$

if the values of their 32 items accessed by the VAMP ISA respectively assembly SPR read functions match.

Isabelle: `VAMPasm2isaSystem/equivanece.sprs_equiv`

Definition 4.4 (Assembly-ISA registers equivalence) Let c_{ASM} and c_{ISA} be assembly and ISA machine configurations. The registers equivalence *r-equiv* holds between the machines if the respective GPR and SPR files are equivalent:

$$\begin{aligned} \textit{r-equiv}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}) \quad &\stackrel{\mathrm{def}}{=} \quad \textit{gpr-equiv}(c_{\mathrm{ASM}}.gpr, c_{\mathrm{ISA}}.gpr) \\ &\wedge \quad \textit{spr-equiv}(c_{\mathrm{ASM}}.spr, c_{\mathrm{ISA}}.spr). \end{aligned}$$

Isabelle: `VAMPasm2isaSystem/equivalence.r_equiv`

Definition 4.5 (Assembly-ISA memory equivalence) Let c_{ASM} and c_{ISA} be assembly and ISA machine configurations. The memory equivalence *m-equiv* holds between the machines if the memories of the machines are equal wordwise taking into consideration data representation:

$$\textit{m-equiv}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \forall\, a < 2^{32} :\ c_{\mathrm{ASM}}.\mathit{Mem}_{\mathrm{ASM}\,\mathit{word}}(a) = [c_{\mathrm{ISA}}.\mathit{Mem}_{\mathrm{ISA}\,\mathit{word}}(a)].$$

Isabelle: `VAMPasm2isaSystem/equivalence.m_equiv`

Altogether, we combine the predicates defined above into a single equivalence relation between VAMP assembly and ISA.

Definition 4.6 (Assembly-ISA equivalence) We call an assembly machine configurations c_{ASM} equivalent to an ISA machine configuration c_{ISA} if the control, the registers, and the memory equivalences hold between them:

$$\begin{aligned} \textit{isa-sim-asm}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=}\ & \textit{c-equiv}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}) \\ \wedge\ & \textit{r-equiv}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}) \\ \wedge\ & \textit{m-equiv}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}). \end{aligned}$$

Isabelle: `VAMPasm2isaSystem/equivalence.equiv_asm_isa`

Adding a conjunct about devices equality we extend the relation to equivalence of combined systems on VAMP assembly and ISA levels.

Definition 4.7 (Assembly-ISA equivalence of combined systems) We call a combined assembly system configuration $c_{\mathrm{ASM+DS}}$ equivalent to a combined ISA system configuration $c_{\mathrm{ISA+DS}}$ if the processor components of the systems are equivalent and the devices components are equal:

$$\begin{aligned} \textit{isa-sim-asm}_{\mathrm{DS}}(c_{\mathrm{ASM+DS}}, c_{\mathrm{ISA+DS}}) \stackrel{\mathrm{def}}{=}\ & \\ & \textit{isa-sim-asm}(c_{\mathrm{ASM+DS}}.\mathit{cpu}, c_{\mathrm{ISA+DS}}.\mathit{cpu}) \\ \wedge\ & c_{\mathrm{ASM+DS}}.\mathit{devs} = c_{\mathrm{ISA+DS}}.\mathit{devs}. \end{aligned}$$

Isabelle: `VAMPasm2isaSystem/equivalence_dev.equiv_asm_isa_with_instr_dev`

4.1.2 Preconditions to Simulation

In order to succeed with assembly executions that could be simulated by ISA computations, a number of preconditions have to be satisfied by assembly programs. These preconditions are defined as predicates over an assembly configuration c_{ASM} and a program given by its start address *addr* in the assembly memory and length *len*. The preconditions are divided into static and dynamic properties.

Static properties. Static properties could be verified in the initial state without any execution of the assembly model. It is necessary that the assembly program fits in

memory of the assembly machine and that the program specified by *addr* and *len* is decodable:

$$\begin{aligned} \textit{stat-prop}(c_{\text{ASM}}, \textit{addr}, \textit{len}) \quad &\stackrel{\text{def}}{=} \quad \textit{addr} \bmod 4 = 0 \\ &\wedge \quad \textit{is-mem-addr}(\textit{addr} + 4 \cdot \textit{len} - 4) \\ &\wedge \quad \textit{decodable-}\pi(\textit{get-data}(c_{\text{ASM}}.m, \textit{addr}, \textit{len})). \end{aligned}$$

Step properties. First, we require that conditions about absence of self-modification code and interrupts hold at every step of the assembly program execution:

$$\begin{aligned} \textit{no-self-mod}(c_{\text{ASM}}, \textit{addr}, \textit{len}) \stackrel{\text{def}}{=} \textit{is-store}(\textit{instr}(c_{\text{ASM}})) \longrightarrow \\ \neg(\textit{addr} \leq \textit{ls-target}(c_{\text{ASM}}) \wedge \textit{ls-target}(c_{\text{ASM}}) < \textit{addr} + 4 \cdot \textit{len}), \end{aligned}$$

$$\textit{no-imal}(c_{\text{ASM}}) \stackrel{\text{def}}{=} c_{\text{ASM}}.\textit{dpc} \bmod 4 = 0,$$

$$\begin{aligned} \textit{no-dmal}(c_{\text{ASM}}) \stackrel{\text{def}}{=} \textit{is-ls-w}(\textit{instr}(c_{\text{ASM}})) \longrightarrow \textit{ls-target}(c_{\text{ASM}}) \bmod 4 = 0 \\ \wedge\, \textit{is-ls-hw}(\textit{instr}(c_{\text{ASM}})) \longrightarrow \textit{ls-target}(c_{\text{ASM}}) \bmod 2 = 0. \end{aligned}$$

Next, it must be guaranteed that we always execute an instruction from the program. This is claimed by the predicate

$$\textit{dpc-in-}\pi(c_{\text{ASM}}, \textit{addr}, \textit{len}) \stackrel{\text{def}}{=} \textit{addr} \leq c_{\text{ASM}}.\textit{dpc} < \textit{addr} + 4 \cdot \textit{len}.$$

Finally, we do not have assembly instructions forbidden by the semantics:

$$\textit{no-trap-rfe}(c_{\text{ASM}}) \stackrel{\text{def}}{=} \textit{instr}(c_{\text{ASM}}) \neq \texttt{rfe} \wedge \textit{instr}(c_{\text{ASM}}) \neq \texttt{trap}(\textit{imm}).$$

Altogether, step properties are:

$$\begin{aligned} \textit{step-prop}(c_{\text{ASM}}, \textit{addr}, \textit{len}) \quad &\stackrel{\text{def}}{=} \quad \textit{is-sys-exec}_{\text{ASM}}(c_{\text{ASM}}) \\ &\wedge \quad \textit{no-self-mod}(c_{\text{ASM}}, \textit{addr}, \textit{len}) \\ &\wedge \quad \textit{no-imal}(c_{\text{ASM}}) \\ &\wedge \quad \textit{no-dmal}(c_{\text{ASM}}) \\ &\wedge \quad \textit{dpc-in-}\pi(c_{\text{ASM}}, \textit{addr}, \textit{len}) \\ &\wedge \quad \textit{no-trap-rfe}(c_{\text{ASM}}). \end{aligned}$$

4.1.3 Simulation without Devices Access

The theorem about simulation of an assembly machine without devices by an ISA machine with devices is used to reason about assembly programs not accessing devices.

The assembly dynamic properties for this theorem are:

$$\begin{aligned} \textit{dyn-prop}(c_{\text{ASM}}, \textit{addr}, \textit{len}, n) \stackrel{\text{def}}{=} \forall\, i < n : \quad \textit{step-prop}(\delta^i_{\text{ASM}}(c_{\text{ASM}}), \textit{addr}, \textit{len}) \\ \wedge\ \textit{no-dev-touch-step}(\delta^i_{\text{ASM}}(c_{\text{ASM}})), \end{aligned}$$

where the predicate

$$\textit{no-dev-touch-step}(c_{\text{ASM}}) \stackrel{\text{def}}{=} \textit{is-ls}(\textit{instr}(c_{\text{ASM}})) \longrightarrow \textit{is-mem-addr}(\textit{ls-target}(c_{\text{ASM}}))$$

requires the target address of a load/store instruction to lie within the normal processor memory whenever such instruction is executed.

Ultimately, the assembly initial conditions are defined as follows:

$$\begin{array}{lcl} \textit{asm-init-cond}(c_{\mathrm{ASM}}, \textit{addr}, \textit{len}, n) & \stackrel{\mathrm{def}}{=} & \textit{stat-prop}(c_{\mathrm{ASM}}, \textit{addr}, \textit{len}) \\ & \wedge & \textit{dyn-prop}(c_{\mathrm{ASM}}, \textit{addr}, \textit{len}, n). \end{array}$$

Having the initial conditions defined, we can state and prove the correctness theorem for this case.

Theorem 4.8 (Assembly-ISA simulation without devices access) Let c_{ASM} be a configuration of an assembly machine which executes in n steps a program that occupies *len* words of memory starting at the address *addr*. Let $c_{\mathrm{ISA+DS}}$ be a configuration of an ISA combined system, and let *seq* be its execution sequence. Provided that (i) both ISA and assembly configurations are valid, (ii) the execution sequence is live and welltyped, (iii) the initial conditions for assembly execution are fulfilled, and (iv) the assembly-ISA equivalence holds, there exists the resulting configuration of the combined ISA system $c'_{\mathrm{ISA+DS}}$ achieved in T steps, such that the assembly-ISA equivalence is preserved and the devices non-interference holds:

$$\begin{array}{lll} & \textit{isa}\surd(c_{\mathrm{ISA+DS}}.\textit{cpu}) & \\ \wedge & \textit{asm}\surd(c_{\mathrm{ASM}}) & \\ \wedge & \textit{seq}\surd(\textit{seq}) & \\ \wedge & \textit{asm-init-cond}(c_{\mathrm{ASM}}, \textit{addr}, \textit{len}, n) & \\ \wedge & \textit{isa-sim-asm}(c_{\mathrm{ASM}}, c_{\mathrm{ISA+DS}}.\textit{cpu}) & \\ \longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{ASM}} : & \\ & \quad \delta^{T}_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \textit{seq}) = c'_{\mathrm{ISA+DS}} & \\ & \wedge \quad \delta^{n}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) = c'_{\mathrm{ASM}} & \\ & \wedge \quad \textit{isa}\surd(c'_{\mathrm{ISA+DS}}.\textit{cpu}) & \\ & \wedge \quad \textit{asm}\surd(c'_{\mathrm{ASM}}) & \\ & \wedge \quad \textit{isa-sim-asm}(c'_{\mathrm{ASM}}, c'_{\mathrm{ISA+DS}}.\textit{cpu}) & \\ & \wedge \quad \textit{non-interf-dev}(c_{\mathrm{ISA+DS}}.\textit{devs}, c'_{\mathrm{ISA+DS}}.\textit{devs}, \textit{seq}, T). & \end{array}$$

Isabelle: `VAMPasm2isaSystem/correctness_isa_dev.asm_wo_dev_simulates_isa_plus_dev`

Proof. For the proof we choose such number of VAMP ISA with devices steps that the execution sequence of that length contains as many processor steps as the assembly machine has made:

$$\textit{proc-steps}(\textit{seq}, T) \;=\; n.$$

This is because one VAMP ISA step is equivalent one VAMP assembly step.

For induction step we consider three cases of the $\delta_{\mathrm{ISA+DS}}$ function (cf. Section 3.4.2). In case that the current sequence element is of form $s = \mathsf{Dev}(\textit{did}, \textit{eifi})$ the devices system makes an independent external step for the device *did*. We accumulate such steps to show that devices do not interfere with the processor or other devices (expressed by *non-interf-dev*). If it is the processor's turn to make a step, $s = \mathsf{Proc}$, we need to show that device access does not take place. We use for that the *no-dev-touch-step* property

and project it to the ISA level. Thus, the combined system transition function $\delta_{\mathrm{ISA+DS}}$ boils down to the normal VAMP ISA transition function δ_{ISA}.

For the processor step we first check for possible interrupts. The absence of interrupts on the ISA level is proven using step properties *step-prop* and the equivalence between VAMP ISA and VAMP assembly. At the end, we show the instruction semantics equivalence for one step:

$$\mathit{isa\text{-}sim\text{-}asm}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}) \longrightarrow \mathit{isa\text{-}sim\text{-}asm}(\delta_{\mathrm{ASM}}(c_{\mathrm{ASM}}), \delta^{\mathrm{woi}}_{\mathrm{ISA}}(c_{\mathrm{ISA}})).$$

This is done for each instruction independently. The most complicated cases here are load and store instructions. The difficulty here is that the memory in VAMP ISA is modeled in the form that we can read or write always two bit vectors of length 32 simultaneously. The decision which data should be read or written is taken separately for each of 8 bytes that constitute these two words. The memory model in the assembly machine is more friendly and features word addressing. □

4.1.4 Simulation with Devices Access

The theorem about simulation of an assembly combined system by an ISA combined system is shown by reasoning about correctness of assembly programs which communicate with devices. It is intended to used for verification of devices interacting primitives.

The assembly combined systems dynamic properties for this theorem are:

$$\begin{aligned}\mathit{dyn\text{-}prop}_{\mathrm{DS}}(c_{\mathrm{ASM+DS}}, \mathit{addr}, \mathit{len}, \mathit{seq}, n) \stackrel{\mathrm{def}}{=}& \\ \forall\, i < n: \quad & \mathit{step\text{-}prop}(\delta^i_{\mathrm{ASM+DS}}(c_{\mathrm{ASM+DS}}, \mathit{seq}).\mathit{cpu}, \mathit{addr}, \mathit{len}) \\ \wedge \quad & \mathit{dev\text{-}touch\text{-}step}(\delta^i_{\mathrm{ASM+DS}}(c_{\mathrm{ASM+DS}}, \mathit{seq}).\mathit{cpu}),\end{aligned}$$

where the predicate

$$\begin{aligned}&\mathit{dev\text{-}touch\text{-}step}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} \\ &\quad \mathit{is\text{-}ls}(\mathit{instr}(c_{\mathrm{ASM}})) \wedge \mathit{is\text{-}dev\text{-}addr}(\mathit{ls\text{-}target}(c_{\mathrm{ASM}})) \longrightarrow \mathit{is\text{-}ls\text{-}w}(\mathit{instr}(c_{\mathrm{ASM}}))\end{aligned}$$

demands that devices are accessed wordwisely.

Thus, the assembly combined system initial conditions are:

$$\begin{aligned}\mathit{asm\text{-}init\text{-}cond}_{\mathrm{DS}}(c_{\mathrm{ASM+DS}}, \mathit{addr}, \mathit{len}, \mathit{seq}, n) \stackrel{\mathrm{def}}{=}& \quad \mathit{stat\text{-}prop}(c_{\mathrm{ASM+DS}}.\mathit{cpu}, \mathit{addr}, \mathit{len}) \\ \wedge& \quad \mathit{dyn\text{-}prop}_{\mathrm{DS}}(c_{\mathrm{ASM+DS}}, \mathit{addr}, \mathit{len}, \mathit{seq}, n).\end{aligned}$$

We continue with the simulation theorem for this case.

Theorem 4.9 (Assembly-ISA simulation with devices access) Let $c_{\mathrm{ASM+DS}}$ be a configuration of an assembly combined system which executes in n steps a program that occupies *len* words of memory starting at the address *addr*. Let $c_{\mathrm{ISA+DS}}$ be a configuration of an ISA combined system, and let *seq* be their execution sequence. Provided that (i) both ISA and assembly configurations are valid, (ii) the execution sequence is live and welltyped, (iii) the initial conditions for assembly combined execution are fulfilled, and (iv) the assembly-ISA equivalence of combined system holds, there exists the resulting configuration of the combined ISA system $c'_{\mathrm{ISA+DS}}$ achieved in n steps, such that the

assembly-ISA equivalence of combined system is preserved:

$$\begin{array}{lll} & isa\surd(c_{\mathrm{ISA+DS}}.cpu) \\ \wedge & asm\surd(c_{\mathrm{ASM+DS}}.cpu) \\ \wedge & seq\surd(seq, c_{\mathrm{ISA+DS}}.devs) \\ \wedge & \textit{asm-init-cond}_{\mathrm{DS}}(c_{\mathrm{ASM+DS}}, addr, len, seq, n) \\ \wedge & \textit{isa-sim-asm}_{\mathrm{DS}}(c_{\mathrm{ASM+DS}}, c_{\mathrm{ISA+DS}}) \\ \longrightarrow & \exists\, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{ASM+DS}} : \\ & \quad \delta^n_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, seq) = c'_{\mathrm{ISA+DS}} \\ & \wedge \quad \delta^n_{\mathrm{ASM+DS}}(c_{\mathrm{ASM+DS}}, seq) = c'_{\mathrm{ASM+DS}} \\ & \wedge \quad isa\surd(c'_{\mathrm{ISA+DS}}.cpu) \\ & \wedge \quad asm\surd(c'_{\mathrm{ASM+DS}}.cpu) \\ & \wedge \quad \textit{isa-sim-asm}_{\mathrm{DS}}(c'_{\mathrm{ASM+DS}}, c'_{\mathrm{ISA+DS}}). \end{array}$$

Isabelle: `VAMPasm2isaSystem/correctness_dev.asm_simulates_isa_dev`

Proof. This theorem is easier than the previous one since we have the same devices systems on both levels and, therefore we deal with a one-to-one simulation. The essential subgoals to be proven here is the equivalence of the memory interfaces in bit vector and natural number representations. □

4.2 Simulation of C0 by VAMP Assembly

This section is copied to a large extend from Leinenbach's thesis.

4.2.1 Memory Layout

A sketch of the memory layout of the C0 compiler in the memory of a VAMP assembly machine in depicted in Figure 4.1. The memory map of the C0 machine starts with the assembly code of the compiled program ([66, Definition 7.36]):

$$code_{\mathrm{prog}}(te, ft, gst(c_{\mathrm{C0}}.mem)).$$

The compiler inserts the initial code at the beginning of the compiled code (cf. [66, Section 7.5.1]). It is necessary for user programs. However, we will use the compiler for the kernel code and define the initialization code ourselves. Hence, in this version of the compiler $code\_init(te, gst, ft) = []$.

The code starts at the address `PROGBASE`, which is a parameter to the C0 compiler and should be aligned by 4, and occupies

$$csize_{\mathrm{prog}}(te, gst(c_{\mathrm{C0}}.mem), ft) \stackrel{\mathrm{def}}{=} |code_{\mathrm{prog}}(te, ft, gst(c_{\mathrm{C0}}.mem))|$$

words ([66, Definition 7.5]). Behind the unused area of size $\mathtt{BUBBLE}_{\mathrm{code}}$ follows the global memory frame starting at address ([66, Definition 7.15]):

$$\begin{array}{lll} \mathtt{ABASE}_{\mathrm{gm}} & \stackrel{\mathrm{def}}{=} & abase_{\mathrm{gm}}(te, ft, gst(c_{\mathrm{C0}}.mem)) \\ & \stackrel{\mathrm{def}}{=} & \mathtt{PROGBASE} + 4 \cdot csize_{\mathrm{prog}}(te, gst(c_{\mathrm{C0}}.mem), ft) + \mathtt{BUBBLE}_{\mathrm{code}}. \end{array}$$

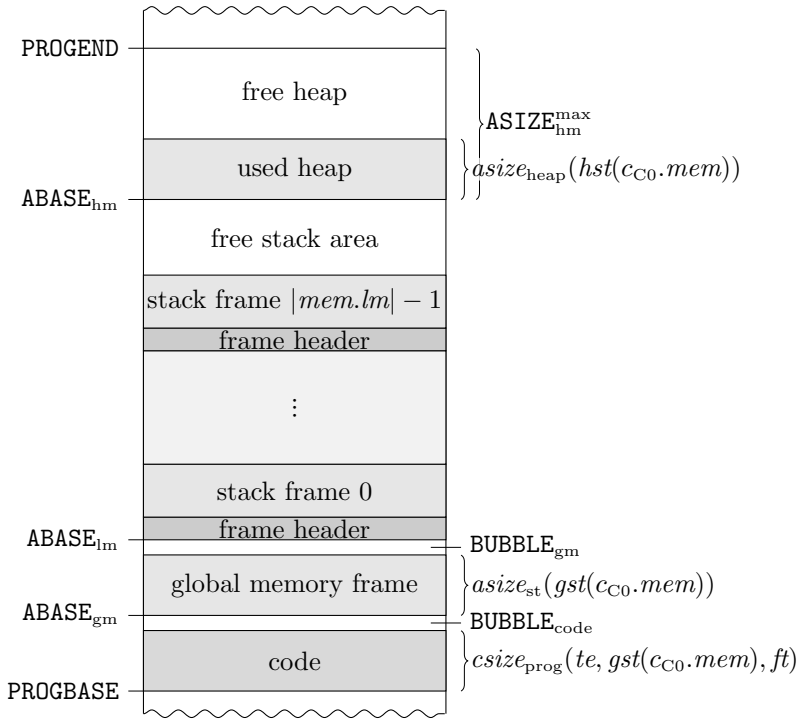

Figure 4.1: Memory layout of the C0 compiler.

The global memory is followed by an unused area of size $\texttt{BUBBLE}_{\text{gm}}$. The stack of local memories starts at address ([66, Definition 7.16]):

$$\begin{aligned}\texttt{ABASE}_{\text{lm}} &\stackrel{\text{def}}{=} abase_{\text{lm}}(te, ft, gst(c_{\text{C0}}.mem))\\ &\stackrel{\text{def}}{=} abase_{\text{gm}}(te, ft, gst(c_{\text{C0}}.mem)) + asize_{\text{st}}(gst(c_{\text{C0}}.mem)) + \texttt{BUBBLE}_{\text{gm}}\end{aligned}$$

and grows to the top. The last part of the memory is the heap. The heap starts at address $\texttt{ABASE}_{\text{hm}}$, which is also a constant parameter to the compiler, and grows to the top as well. The size of the heap memory is computed by the function $asize_{\text{heap}}(hst(c_{\text{C0}}.mem))$ ([66, Definition 7.12]) and is bounded by the constant $\texttt{ASIZE}^{\max}_{\text{hm}}$. The first free address behind the program is

$$\texttt{PROGEND} \;=\; \texttt{ABASE}_{\text{hm}} + \texttt{ASIZE}^{\max}_{\text{hm}}.$$

4.2.2 Simulation Relation

The simulation relation for C0 is parameterized with an allocation function of type

$$Alloc \stackrel{\text{def}}{=} Gvar \mapsto (\mathbb{N} \times \mathbb{N}).$$

The allocation function maps g-variables g to pairs $alloc(g) = (b, s)$ of the allocated base address b for g and the allocated size s of g's type.

The simulation relation $consis :: Tenv \times Functable \times C_{\text{C0}} \times Alloc \times C_{\text{ASM}} \mapsto \mathbb{B}$ defines whether a C0 configuration and an assembly configuration are equivalent with respect

to a given allocation function. It is defined as a conjunction of different individual consistency properties stated below.

Code consistency. Let te be a type environment, ft a function table, c_{C0} a C0 configuration, and c_{ASM} a configuration of the VAMP assembly machine. The code consistency ([66, Definition 8.12]):

$$\mathit{consis}_{\mathrm{code}}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}})$$

requires that the compiled code of the program $\mathit{code}_{\mathrm{prog}}(te, ft, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}))$ is stored at address `PROGBASE` in the assembly configuration.

Control consistency. Let te be a type environment, ft be a function table, c_{C0} a C0 configuration, and c_{ASM} a configuration of the VAMP assembly machine. The control consistency ([66, Definitions 8.13, 8.14]):

$$\mathit{consis}_{\mathrm{c}}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}})$$

requires that the program counters of the assembly machine point to the start address of the code which has been generated for the head of the current program rest, in case it is not empty, and that return addresses of all stack frames point directly behind the code of the function call statements which generated these stack frames.

Allocation consistency. Let te be a type environment, ft a function table, c_{C0} a C0 configuration, and *alloc* an allocation function. The allocation consistency ([66, Definitions 8.15, 8.16, 8.17]):

$$\mathit{consis}_{\mathrm{alloc}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc})$$

requires that the allocation function returns for all valid g-variables g meaningful values: the first component returns the same values as $\mathit{abase}_{\mathrm{g}}(te, ft, \mathit{sc}(c_{\mathrm{C0}}.\mathit{mem}), g)$ (cf. [66, Definition 7.17]) for these variables, and the second component returns their allocated sizes.

Value consistency. Let c_{C0} be a C0 configuration, *alloc* an allocation function, and c_{ASM} a configuration of the VAMP assembly machine. The value consistency ([66, Definitions 8.18, 8.19]):

$$\mathit{consis}_{\mathrm{v}}(c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}})$$

requires that for all reachable non-pointer g-variables g of elementary type their values are properly represented in the assembly configuration. This is done by means of the function *vmatch*, which compares values in the *C0*configuration $\mathit{value}_{\mathrm{g}}(c_{\mathrm{C0}}.\mathit{mem}, g)$ and in the VAMP assembly machine $c_{\mathrm{ASM}}.m_{\mathit{word}}(\mathit{alloc}(g).b)$.

Pointer consistency. Let c_{C0} be a C0 configuration, *alloc* an allocation function, and c_{ASM} a configuration of the VAMP assembly machine. The pointer consistency ([66, Definitions 8.20, 8.21]):

$$\mathit{consis}_{\mathrm{p}}(c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}})$$

is similar to value consistency but a different notion of *equality* is used: $\mathit{vmatch}_{\mathrm{ptr}}$.

Register consistency. Let te be a type environment, ft a function table, c_{C0} a C0 configuration, *alloc* an allocation function, and c_{ASM} a configuration of the VAMP assembly machine. The register consistency ([66, Definition 8.22]):

$$\mathit{consis}_{\mathrm{r}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}})$$

argues about the content of some special registers: register r_{sbase} stores the base address of the global memory frame $\mathit{abase}_{\mathrm{gm}}(te, ft, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}))$, register r_{htop} stores the first unused address on the heap $\mathtt{ABASE}_{\mathrm{hm}} + \mathit{asize}_{\mathrm{heap}}(\mathit{hst}(c_{\mathrm{C0}}.\mathit{mem}))$, and register r_{lframe} stores the base address of the current (top) stack frame $\mathit{abase}_{\mathrm{lm}}(te, ft, \mathit{sc}(c_{\mathrm{C0}}.\mathit{mem}), |c_{\mathrm{C0}}.\mathit{mem}.\mathit{lm}| - 1)$.

Frame header consistency. Let te be a type environment, ft a function table, c_{C0} a C0 configuration, *alloc* an allocation function, and c_{ASM} a configuration of the VAMP assembly machine. The frame header consistency ([66, Definition 8.23]):

$$\mathit{consis}_{\mathrm{fh}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}})$$

requires that for all i the previous stack pointer in the frame header of the i-th local stack frame contains the allocated base address of the frame $i-1$: $\mathit{abase}_{\mathrm{lm}}(te, ft, \mathit{sc}(c_{\mathrm{C0}}.\mathit{mem}), i-1)$ and that the return destination contains the allocated base address of the i-th return destination $\mathit{alloc}(c_{\mathrm{C0}}.\mathit{mem}.\mathit{lm}[\![i]\!].\mathit{res}).b$.

Altogether. The data consistency is defined as follows ([66, Definition 8.24]):

$$\begin{aligned}
\mathit{consis}_{\mathrm{d}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=}\ & \mathit{consis}_{\mathrm{alloc}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}) \\
\wedge\ & \mathit{consis}_{\mathrm{v}}(c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{p}}(c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{r}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{fh}}(te, ft, \mathit{alloc}, c_{\mathrm{ASM}})
\end{aligned}$$

Ultimately, the C0 consistency is ([66, Definition 8.11]):

$$\begin{aligned}
\mathit{consis}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=}\ & \mathit{consis}_{\mathrm{code}}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{c}}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{d}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}).
\end{aligned}$$

4.2.3 Resources Restriction

Besides the requirement that the compiled code fits into memory an important proof goal of the compiler correctness proof will be that during execution the compiled program does not use more memory than it is available on the target machine. The following predicates check this.

Enough stack available. Let te be a type environment, ft a function table, and c_{C0} a C0 configuration. The predicate ([66, Definition 8.25])

$$\mathit{avail}_{\mathrm{stack}}(te, ft, c_{\mathrm{C0}})$$

tests whether the topmost stack frame ends below the heap base:

$$\mathit{abase}_{\mathrm{lm}}(te, ft, \mathit{sc}(c_{\mathrm{C0}}.\mathit{mem}), |c_{\mathrm{C0}}.\mathit{mem}.\mathit{lm}|) \ \leq\ \mathtt{ABASE}_{\mathrm{hm}}.$$

Enough heap available. Let c_{C0} be a C0 configuration and let $addr_{\max}$ denote the maximum address in the program's address space. The predicate ([66, Definition 8.26])

$$avail_{\mathrm{heap}}(c_{\mathrm{C0}})$$

tests whether the allocated heap size is appropriately bounded:

$$asize_{\mathrm{heap}}(hst(c_{\mathrm{C0}}.mem)) \ \leq \ \mathtt{ASIZE}_{\mathrm{hm}}^{\max}.$$

Sufficient memory. Both predicates are combined into the sufficient memory available predicate ([66, Definition 8.26]):

$$\begin{array}{rcl} avail_{\mathrm{mem}}(te, ft, c_{\mathrm{C0}}) & \stackrel{\mathrm{def}}{=} & avail_{\mathrm{stack}}(te, ft, c_{\mathrm{C0}}) \\ & \wedge & avail_{\mathrm{heap}}(c_{\mathrm{C0}}). \end{array}$$

4.2.4 Dynamic Properties

The necessary dynamic properties are the absence of assembly statements and the sufficient memory requirement:

$$\begin{array}{rl} \textit{dyn-C0-props}(te, ft, c_{\mathrm{C0}}, n) \stackrel{\mathrm{def}}{=} & \\ \forall\, i < n:\ \delta^i_{\mathrm{C0}}(te, ft, c_{\mathrm{C0}}) = \lfloor c^i_{\mathrm{C0}} \rfloor \ \longrightarrow\ \neg \textit{is-asm}(\textit{stmt}(c^i_{\mathrm{C0}})) & \\ \wedge\quad \forall\, i \leq n:\ \delta^i_{\mathrm{C0}}(te, ft, c_{\mathrm{C0}}) = \lfloor c^i_{\mathrm{C0}} \rfloor \ \longrightarrow\ avail_{\mathrm{mem}}(te, ft, c^i_{\mathrm{C0}}). & \end{array}$$

4.2.5 Assembly Execution Properties

Leinenbach uses at this place slightly different definitions. Essentially, they are similar to those that are used in the section 4.1 for assembly correctness. Due to simplicity we omit equivalence proofs between them. Only one definition has to be additionally introduced. It guarantees that all changes are done in some given memory range:

$$\textit{accessed-range} :: C_{\mathrm{ASM}} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B},$$

$$\begin{array}{rl} \textit{accessed-range}(c_{\mathrm{ASM}}, start_{\mathrm{data}}, end_{\mathrm{data}}) \stackrel{\mathrm{def}}{=} & \\ \quad \textit{is-ls}(\textit{instr}(c_{\mathrm{ASM}})) & \\ \quad \longrightarrow \quad start_{\mathrm{data}} \leq \textit{ls-target}(c_{\mathrm{ASM}}) & \\ \qquad \wedge\ \textit{ls-target}(c_{\mathrm{ASM}}) + \textit{ls-width}(c_{\mathrm{ASM}}) \leq end_{\mathrm{data}}. & \end{array}$$

Ultimately, the predicate

$$\textit{asm-exec-props} :: C_{\mathrm{ASM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B}$$

holds if n VAMP assembly steps starting from configuration c_{ASM} would not generate interrupts on the ISA level. Here the code range of an assembly program is given by the start address $base_{\mathrm{code}}$ and its size len_{code}. The range of addresses which the program

accesses are given by its start address $start_{\text{data}}$ and end address end_{data}:

$$\begin{aligned}
&\textit{asm-exec-props}(c_{\text{ASM}}, \textit{base}_{\text{code}}, \textit{len}_{\text{code}}, \textit{start}_{\text{data}}, \textit{end}_{\text{data}}, n) \stackrel{\text{def}}{=} \\
&\qquad \forall\, i < n: \quad \textit{no-self-mod}(\delta^i_{\text{ASM}}(c_{\text{ASM}}), \textit{base}_{\text{code}}, \textit{len}_{\text{code}}) \\
&\qquad\qquad \wedge \quad \textit{no-imal}(\delta^i_{\text{ASM}}(c_{\text{ASM}})) \\
&\qquad\qquad \wedge \quad \textit{no-dmal}(\delta^i_{\text{ASM}}(c_{\text{ASM}})) \\
&\qquad\qquad \wedge \quad \textit{dpc-in-}\pi(\delta^i_{\text{ASM}}(c_{\text{ASM}}), \textit{base}_{\text{code}}, \textit{len}_{\text{code}}) \\
&\qquad\qquad \wedge \quad \textit{no-trap-rfe}(\delta^i_{\text{ASM}}(c_{\text{ASM}})) \\
&\qquad\qquad \wedge \quad \delta^{i+1}_{\text{ASM}}(c_{\text{ASM}}).\textit{spr} = c_{\text{ASM}}.\textit{spr} \\
&\qquad\qquad \wedge \quad \textit{accessed-range}(\delta^i_{\text{ASM}}(c_{\text{ASM}}), \textit{start}_{\text{data}}, \textit{end}_{\text{data}}).
\end{aligned}$$

4.2.6 Simulation Theorem

Now we present the top-level correctness theorem of the C0 compiler.

Theorem 4.10 (C0-assembly simulation) Assume that (i) the initial C0 program is translatable, (ii) the current valid C0 configuration c_{C0} and some valid assembly machine c_{ASM} are consistent with respect to an allocation function *alloc*, (iii) the C0 computation starting from this configuration, does not produce a *None* configuration till the step n, and (iv) during these n steps we execute only non-assembly statements having enough stack and heap memory, then there exists a number of VAMP assembly model steps T, such that by executing this number of steps starting from the given assembly configuration we transit into a new valid configuration c'_{ASM}. Moreover, there exists a new allocation function *alloc*′, such that (i) the C0 machine after n steps c'_{C0} is valid, (ii) the consistency relation holds, and (iii) the execution of the compiled code of the executed statements will not produce any interrupts. Formally:

$$\begin{aligned}
&(te, ft, \textit{gst}(c_{\text{C0}}.\textit{mem})) \in \textit{xltbl}_{\text{prog}} \\
\wedge \quad &\textit{consis}(te, ft, c_{\text{C0}}, \textit{alloc}, c_{\text{ASM}}) \\
\wedge \quad &\textit{asm}\surd(c_{\text{ASM}}) \\
\wedge \quad &c_{\text{C0}} \in C0'\surd(te, ft) \\
\wedge \quad &\delta^n_{\text{C0}}(te, ft, c_{\text{C0}}) = \lfloor c'_{\text{C0}} \rfloor \\
\wedge \quad &\textit{dyn-C0-props}(te, ft, c_{\text{C0}}, n) \\
\longrightarrow \quad &\exists\, T, \textit{alloc}', c'_{\text{ASM}}: \\
&\qquad \delta^T_{\text{ASM}}(c_{\text{ASM}}) = c'_{\text{ASM}} \\
&\qquad \wedge \quad \textit{asm}\surd(c'_{\text{ASM}}) \\
&\qquad \wedge \quad c'_{\text{C0}} \in C0'\surd(te, ft) \\
&\qquad \wedge \quad \textit{consis}(te, ft, c'_{\text{C0}}, \textit{alloc}', c'_{\text{ASM}}) \\
&\qquad \wedge \quad \textit{asm-exec-props}(c_{\text{ASM}}, \texttt{PROGBASE}, \\
&\qquad\qquad\qquad\qquad \textit{csize}_{\text{prog}}(te, \textit{gst}(c_{\text{C0}}.\textit{mem}), ft), \\
&\qquad\qquad\qquad\qquad \textit{abase}_{\text{gm}}(te, ft, \textit{gst}(c_{\text{C0}}.\textit{mem})), \texttt{PROGEND}, T).
\end{aligned}$$

Isabelle: `C0SS2VAMPisaSystem/C0SS2VAMPasm.c0_compiler_correct`

The theorem is proven by induction on n using the theorem for correctness of the compiler's induction step and many other lemmas from [66].

4.3 Simulation of C0 by VAMP ISA

In this section we combine the compiler correctness theorem with the equivalence theorem between VAMP assembly and ISA. As a result we obtain an overall correctness theorem of a *ministack* — the system comprising three semantical layers. By this we get a number of benefits: in many places we will apply only one theorem instead of two as well as save effort on proving one theorem's assumptions which follow from conclusions of the other.

4.3.1 Simulation Relation

The new simulation relation represents transitivity of the VAMPs equivalence and the compiler consistency. We hide the compiler allocation function inside the ministack relation since the range of its values could be computed from the C0 configuration by the function $abase_{\mathrm{g}}$ (cf. *allocation consistency*):

$$\textit{C0-sim-isa} :: \textit{Tenv} \times \textit{Functable} \times C_{\mathrm{C0}} \times C_{\mathrm{ISA}} \mapsto \mathbb{B},$$

$$\begin{aligned}\textit{C0-sim-isa}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \exists\, c_{\mathrm{ASM}}, \textit{alloc} :\ & \textit{asm}\surd(c_{\mathrm{ASM}}) \\ \wedge\ & \textit{consis}(te, ft, c_{\mathrm{C0}}, \textit{alloc}, c_{\mathrm{ASM}}) \\ \wedge\ & \textit{isa-sim-asm}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}).\end{aligned}$$

4.3.2 Additional Conclusions

In order to effectively apply the simulation theorem between C0 and VAMP ISA in the context of systems verification we extend the theorem's conclusion with a number of additional properties. Basically, they state that certain components of the system stay unchanged during the C0 execution. For this we have to define two additional predicates.

First of all we introduce a predicate which claims that no special registers are modified:

$$\textit{no-mod-spr} :: \textit{Regf}_{\mathrm{ISA}} \times \textit{Regf}_{\mathrm{ISA}} \mapsto \mathbb{B},$$

$$\textit{no-mod-spr}(spr, spr') \stackrel{\mathrm{def}}{=} \forall\, r \in \textit{sprs}_{\mathrm{ISA}} :\ spr'(r) = spr(r).$$

Second, the following predicate states that only the part of the memory between the addresses a_{begin} and a_{end} is modified:

$$\textit{only-mod-mem} :: \textit{Mem}_{\mathrm{ISA}} \times \textit{Mem}_{\mathrm{ISA}} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B},$$

$$\begin{aligned}&\textit{only-mod-mem}(m,m', a_{\mathrm{begin}}, a_{\mathrm{end}}) \stackrel{\mathrm{def}}{=} \\ &\qquad \forall\, a :\ a < a_{\mathrm{begin}} \vee a_{\mathrm{end}} \leq a \ \longrightarrow\ {m'}_{word}(a) = m_{word}(a).\end{aligned}$$

4.3.3 Simulation Theorem

The theorem about simulation between the C0 and VAMP ISA levels contains all assumptions from the C0-assembly simulation as well as restriction on the execution sequence for the external environment. We do not have devices on the C0 level. Hence, we combine compiler correctness with the assembly-ISA simulation without devices access (Theorem 4.8).

Theorem 4.11 (C0-ISA simulation) Assume that (i) the initial C0 program is translatable, (ii) the C0-ISA relation holds for the current valid C0 configuration c_{C0} and some

valid VAMP ISA machine $c_{\text{ISA+DS}}.cpu$ being in the system mode, (iii) the C0 computation starting from this configuration, does not produce a *None* configuration up to the step n, (iv) during these n steps we execute only non-assembly statements having enough stack and heap memory, (v) the last C0 address `PROGEND` does not lie in the devices range, and (vi) the execution sequence is live and welltyped, then there exists a number of ISA with devices steps T during which the ISA combined system transits to a valid resulting state $c'_{\text{ISA+DS}}$. Moreover, for it and a corresponding valid C0 configuration c'_{C0} the following holds: (i) the C0-ISA relation is preserved, (ii) the special purpose registers are unchanged, (iii) only the memory, which belongs to the C0 program is possibly changed, and (iv) the devices non-interference holds. Formally:

$$
\begin{aligned}
& (te, ft, gst(c_{\text{C0}}.mem)) \in xltbl_{\text{prog}} \\
\wedge\ & \textit{C0-sim-isa}(te, ft, c_{\text{C0}}, c_{\text{ISA+DS}}.cpu) \\
\wedge\ & isa\surd(c_{\text{ISA+DS}}.cpu) \\
\wedge\ & c_{\text{C0}} \in C0'\surd(te, ft) \\
\wedge\ & \textit{is-sys-exec}_{\text{ISA}}(c_{\text{ISA+DS}}.cpu) \\
\wedge\ & \delta^n_{\text{C0}}(te, ft, c_{\text{C0}}) = \lfloor c'_{\text{C0}} \rfloor \\
\wedge\ & \textit{is-mem-addr}(\texttt{PROGEND}) \\
\wedge\ & \textit{dyn-C0-props}(te, ft, c_{\text{C0}}, n) \\
\wedge\ & seq\surd(seq, c_{\text{ISA+DS}}.devs) \\
\longrightarrow\ & \exists\, T, c'_{\text{ISA+DS}} : \\
& \quad \delta^T_{\text{ISA+DS}}(c_{\text{ISA+DS}}, seq) = c'_{\text{ISA+DS}} \\
& \wedge\ isa\surd(c'_{\text{ISA+DS}}.cpu) \\
& \wedge\ c'_{\text{C0}} \in C0'\surd(te, ft) \\
& \wedge\ \textit{C0-sim-isa}(te, ft, c'_{\text{C0}}, c'_{\text{ISA+DS}}.cpu) \\
& \wedge\ \textit{no-mod-spr}(c_{\text{ISA+DS}}.cpu.spr, c'_{\text{ISA+DS}}.cpu.spr) \\
& \wedge\ \textit{only-mod-mem}(c_{\text{ISA+DS}}.cpu.m, c'_{\text{ISA+DS}}.cpu.m, \\
& \qquad\qquad abase_{\text{gm}}(te, ft, gst(c_{\text{C0}}.mem)), \texttt{PROGEND}) \\
& \wedge\ \textit{non-interf-dev}(c_{\text{ISA+DS}}.devs, c'_{\text{ISA+DS}}.devs, seq, T).
\end{aligned}
$$

Isabelle: `COSS2VAMPisaSystem/mini_stack_wo_dev.C0_mini_stack_isa`

Proof. By unfolding the predicate *C0-sim-isa* we obtain some assembly configuration c_{ASM} with the following properties:

$$
\begin{aligned}
& asm\surd(c_{\text{ASM}}) \\
\wedge\ & \textit{consis}(te, ft, c_{\text{C0}}, alloc, c_{\text{ASM}}) \\
\wedge\ & \textit{isa-sim-asm}(c_{\text{ASM}}, c_{\text{ISA+DS}}.cpu).
\end{aligned}
$$

Now we have all necessary assumptions to apply the C0-correctness theorem (Theorem 4.10). After this we have the following: a new assembly configuration c'_{ASM}, an allocation function $alloc'$, and a number of assembly steps T_{ASM} with the following prop-

erties:

$$\begin{aligned}
& \delta_{\mathrm{ASM}}^{T_{\mathrm{ASM}}}(c_{\mathrm{ASM}}) = c'_{\mathrm{ASM}} \\
\wedge\ & \mathit{asm}\surd(c'_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}(\mathit{te}, \mathit{ft}, c'_{\mathrm{C0}}, \mathit{alloc}', c'_{\mathrm{ASM}}) \\
\wedge\ & \mathit{asm\text{-}exec\text{-}props}(c_{\mathrm{ASM}}, \texttt{PROGBASE}, \\
& \qquad\qquad \mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft}), \\
& \qquad\qquad \mathit{abase}_{\mathrm{gm}}(\mathit{te}, \mathit{ft}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem})), \texttt{PROGEND}, T_{\mathrm{ASM}}).
\end{aligned}$$

In order to apply the assembly-ISA simulation theorem without device access (Theorem 4.8) we instantiate the number of steps n with T_{ASM}, the code start address *addr* with PROGBASE, and the code length *len* with $\mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft})$. The only condition we have to show is:

$$\mathit{asm\text{-}init\text{-}cond}(c_{\mathrm{ASM}}, \texttt{PROGBASE}, \mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft}), T_{\mathrm{ASM}}).$$

First, we prove its static properties conjunct:

$$\mathit{stat\text{-}prop}(c_{\mathrm{ASM}}, \texttt{PROGBASE}, \mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft})).$$

Subgoal 1. PROGBASE mod 4: follows from definition of the constant.

Subgoal 2. $\mathit{is\text{-}mem\text{-}addr}(\texttt{PROGBASE} + 4 \cdot \mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft}) - 4)$: from the translatable program predicate we conclude that

$$\begin{aligned}
\texttt{PROGBASE} + 4 \cdot \mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft}) - 4 \quad & < \quad 2^{25} - 4 \\
& < \quad \langle 1^{17} 0^{15} \rangle \\
& = \quad \texttt{DEVICES_BORDER} \\
& \longrightarrow \quad \mathit{is\text{-}mem\text{-}addr}.
\end{aligned}$$

Subgoal 3. $\mathit{decodable}\text{-}\pi(\mathit{get\text{-}data}(c_{\mathrm{ASM}}.m, \texttt{PROGBASE}, \mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft})))$: follows from the code consistency $\mathit{consis}_{\mathrm{code}}$.

Second, we prove the dynamic properties:

$$\mathit{dyn\text{-}prop}(c_{\mathrm{ASM}}, \texttt{PROGBASE}, \mathit{csize}_{\mathrm{prog}}(\mathit{te}, \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}), \mathit{ft}), T_{\mathrm{ASM}}),$$

where for every intermediate assembly configuration $c^i_{\mathrm{ASM}} = \delta^i_{\mathrm{ASM}}(c_{\mathrm{ASM}})$ with $i < T_{\mathrm{ASM}}$ we have to show the following:

Subgoal 4. $\mathit{is\text{-}sys\text{-}exec}_{\mathrm{ASM}}(c^{\mathrm{i}}_{\mathrm{ASM}})$: for $i = 0$ we can show that

$$\begin{aligned}
& \mathit{isa\text{-}sim\text{-}asm}(c_{\mathrm{ASM}}, c_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
& \qquad \longrightarrow\ \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) = \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ISA}}(c_{\mathrm{ISA+DS}}.\mathit{cpu}),
\end{aligned}$$

for $0 < i$ using *asm-exec-props* we have

$$c^{\mathrm{i}}_{\mathrm{ASM}}.\mathit{spr} = c_{\mathrm{ASM}}.\mathit{spr}.$$

Thus, the contents of the mode register and the status register stay the same.

Subgoal 5. *dpc-in-π*($c^{\mathrm{i}}_{\mathrm{ASM}}$, PROGBASE, $csize_{\mathrm{prog}}(te, gst(c_{\mathrm{C0}}.mem), ft)$), *no-dmal*($c^{\mathrm{i}}_{\mathrm{ASM}}$), *no-imal*($c^{\mathrm{i}}_{\mathrm{ASM}}$), *no-self-mod*($c^{\mathrm{i}}_{\mathrm{ASM}}$, PROGBASE, $csize_{\mathrm{prog}}(te, gst(c_{\mathrm{C0}}.mem), ft)$), and *no-trap-rfe*($c^{\mathrm{i}}_{\mathrm{ASM}}$): follow from the predicate *asm-exec-props*.

Subgoal 6. *no-dev-touch-step*($c^{\mathrm{i}}_{\mathrm{ASM}}$): from the predicate *asm-exec-props* we have that

$$\textit{accessed-range}(c^{\mathrm{i}}_{\mathrm{ASM}}, \textit{abase}_{\mathrm{gm}}(te, ft, gst(c_{\mathrm{C0}}.mem)), \texttt{PROGEND}).$$

Unfolding this definition and using the theorem's assumption PROGEND $\leq$ DEVICES_BORDER we have in case of a memory access instruction *is-ls*(*instr*($c^{\mathrm{i}}_{\mathrm{ASM}}$)) the following:

$$\begin{array}{rcl} \textit{ls-target}(c^{\mathrm{i}}_{\mathrm{ASM}}) & \leq & \texttt{PROGEND} - \textit{ls-width}(c^{\mathrm{i}}_{\mathrm{ASM}}) \\ & \leq & \texttt{PROGEND} \\ & < & \texttt{DEVICES\_BORDER} \\ & \longrightarrow & \textit{is-mem-addr}. \end{array}$$

Now we have a new ISA combined system configuration $c'_{\mathrm{ISA+DS}}$ and a number of ISA steps T_{ISA} with the following properties:

$$\begin{array}{ll} & \delta^{T_{\mathrm{ISA}}}_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, seq) = c'_{\mathrm{ISA+DS}} \\ \wedge & \textit{isa}\surd(c'_{\mathrm{ISA+DS}}.cpu) \\ \wedge & \textit{isa-sim-asm}(c'_{\mathrm{ASM}}, c'_{\mathrm{ISA+DS}}.cpu) \\ \wedge & \textit{non-interf-dev}(c_{\mathrm{ISA+DS}}.devs, c'_{\mathrm{ISA+DS}}.devs, seq, T_{\mathrm{ISA}}). \end{array}$$

To show the conclusion of the theorem we instantiate T with T_{ISA}, *alloc*′ with *alloc*′, and $c'_{\mathrm{ISA+DS}}$ with $c'_{\mathrm{ISA+DS}}$. We already have an ISA combined system execution, ISA validity, devices non-interference, and C0-ISA simulation via c'_{ASM}. The equality of special purpose registers is shown as follows:

$$\begin{array}{ll} & \textit{spr-equiv}(c_{\mathrm{ASM}}.spr, c_{\mathrm{ISA+DS}}.cpu.spr) \\ \wedge & \textit{spr-equiv}(c'_{\mathrm{ASM}}.spr, c'_{\mathrm{ISA+DS}}.cpu.spr) \\ \wedge & c'_{\mathrm{ASM}}.spr = c_{\mathrm{ASM}}.spr \\ \longrightarrow & \forall\, r \in sprs_{\mathrm{ISA}} :\ c'_{\mathrm{ISA+DS}}.cpu.spr(r) = c_{\mathrm{ISA+DS}}.cpu.spr(r). \end{array}$$

Finally, the arguments for the conclusion about the memory are as follows:

$$\begin{array}{ll} & \textit{m-equiv}(c_{\mathrm{ASM}}, c_{\mathrm{ISA+DS}}.cpu) \\ \wedge & \textit{m-equiv}(c'_{\mathrm{ASM}}, c'_{\mathrm{ISA+DS}}.cpu) \\ \wedge & \forall\, i < T_{\mathrm{ASM}} :\ \textit{accessed-range}(\delta^{i}_{\mathrm{ASM}}(c_{\mathrm{ASM}}), \\ & \qquad\qquad \textit{abase}_{\mathrm{gm}}(te, ft, gst(c_{\mathrm{C0}}.mem)), \texttt{PROGEND}) \\ \longrightarrow & \textit{only-mod-mem}(c'_{\mathrm{ISA+DS}}.cpu.m, c_{\mathrm{ISA+DS}}.cpu.m, \\ & \qquad\qquad \textit{abase}_{\mathrm{gm}}(te, ft, gst(c_{\mathrm{C0}}.mem)), \texttt{PROGEND}). \end{array}$$

□

Chapter 5

CVM: Communicating Virtual Machines

Contents

In this chapter we discuss how one can formally define an operating system microkernel. We follow the paper-and-pencil model of communicating virtual machines (CVM) introduced in [41]. CVM is a computational model for concurrent user processes interacting with a generic microkernel and devices. CVM is implemented in $C0_A$ as a framework [57] featuring virtual memory [50], demand paging [8], memory management, and low-level inter-process and devices communications [5]. Most of these features are implemented in the form of so called microkernel primitives [106]. Primitives are functions with inline assembly parts realizing basic operations which constitute the kernel's functionality. The framework can be linked on the source code level with an abstract kernel, an interface to users, in order to obtain a concrete kernel, a program that can be translated and run on a target machine, e.g., a VAMP processor. In this chapter we elaborate on details how the CVM framework is formally defined. As CVM is parametrized with an abstract kernel its computations do not depend on particular shapes of abstract kernels. We therefore present only general requirements to it. Two different abstract kernels are used in Verisoft: a general purpose microkernel VAMOS which is inspired by but is not very close to L4, and an OSEKtime-like microkernel OLOS which is used in a distributed automotive real-time system establishing eCall functionality. For details on VAMOS consult the theses of Daum [30] and Dörenbächer [34]. OLOS is described in the thesis of Knapp [64].

We start this chapter by an overview of the CVM model: Section 5.1 presents layers of CVM and describes how the target machine, user processes, and the kernel can be formally defined by the models introduced in Chapter 3. We continue by defining CVM formally. In Section 5.2 we elaborate on the CVM configurations and in Section 5.3 on the formal CVM semantics. The semantics of CVM distinguishes cases of a kernel

and user-processes execution. The case of a kernel execution comprises, among others, a primitive execution. Section 5.4 is devoted to a formal description of the primitives' effects.

5.1 Overview

In this section we give a brief overview of the computational model CVM, its computations and the target architecture.

Layering is a classical approach in computer science to handle complexity of systems design. A monolithic computer system is organized in a number of layers, such that each upper layer is an abstraction of the the one below. Chapter 3 provides a good example: VAMP assembly is an abstraction of VAMP ISA, though, it can be abstracted itself to C0 small-step semantics. Correctness of such abstractions is justified by simulation theorems between adjacent layers — we state such theorems in Chapter 4.

In Verisoft the same layering approach has been taken to formulate and prove a correctness theorem for an operating-system microkernel. Gargano et al. introduce in [41] an abstract parallel model of computation called communicating virtual machines (CVM). This model formalizes concurrent user processes interacting with a kernel and devices. Interleaved executions of user processes and the kernel proceed on an underlying hardware modeled by a VAMP ISA combined system. We refer to this hardware model as the CVM target layer. Next, we elaborate on the CVM target layer and on the layers comprised by CVM itself.

5.1.1 The Target Layer of CVM

The main purpose of a microkernel is to provide multiple users access to shared computational resources like physical memory and devices. Therefore, a particular target hardware model has to be taken into consideration when designing a microkernel or a microkernel framework. The target hardware architecture of a microkernel considered in this thesis is the VAMP ISA with devices. Below, we briefly highlight how this hardware platform allows us to implement some fundamental features of a microkernel.

Physical memory sharing is realized in CVM by memory virtualization: a kernel ensures that each user process has a notion of its own large address space. User processes access memory by virtual addresses which are translated to physical ones by a memory management unit on the hardware side, or by the kernel on the software. We allow address spaces of user processes to exceed real memory of the physical machine. This feature is supported by means of demand paging: we partition available physical memory into small consecutive portions of data, called pages, which are stored either in fast but strongly limited in size *physical memory*, or in large but slower auxiliary memory, called *swap memory*. We store the swap memory on a hard disk. The address translation algorithm of VAMP can determine where a certain page lies. In case the desired pages resides in the physical memory the kernel can provide an immediate access. Otherwise, the page is on the hard disk. The processor signals it by raising a page-fault interrupt. The kernel reacts to this interrupt by transferring the page from the hard disk to the main memory.

Communication of user processes with devices is supported by memory-mapped devices of the VAMP combined system. When a user wants to talk to a device it signals it to the kernel by invoking a special system call. The user specifies an address corresponding to the port of a needed device. The kernel translates this address into a physical one and writes into the part of the physical memory corresponding to the device port. The

transition function of the VAMP ISA with devices model takes care that the written data is transferred to the device.

As the VAMP ISA with devices model represents real physical computational resources we will refer to it further in the thesis as the *physical combined system*. A processor component of this system will be called a *physical machine*.

5.1.2 User Processes

[41] describes virtual machines as a model for user processes. In this paper virtual machines are VAMP ISA processors with uniform virtual memories, no address translation, and undefined interrupt mechanism. Conceptually this abstraction corresponds to the model of VAMP assembly defined in Section 3.2 with the only difference in data representation. However, it is very unlikely that someone prefers to program a user processes on the bit-vector level rather than on an assembly level where data is represented as integer numbers. Thus, we model user processes with the VAMP assembly semantics.

5.1.3 Layers of CVM

CVM provides a microkernel architecture consisting of two layers. The general idea behind this layering is to separate a kernel into two parts: the one that can be purely implemented in a high-level programming language, say C0, and the other that inevitably contains inline assembly code because it provides operations which access hardware registers, devices, and physical memory parts which are not accessibly through C0 variables.

The upper layer, called an abstract kernel, is a C0 program. Its computations are modeled by the C0 small-step semantics. Besides ordinary C0 functions the abstract kernel can call a number of special functions, called CVM primitives. These functions have no implementation within the abstract kernel, and are therefore called externally, e.g., by means of the *esCall* statement. CVM primitives can alter states of user processes as well as target-hardware registers and devices. They implement basic means needed for a microkernel programmer: copy data between processes, manage size of virtual memory given to processes, send data to devices, etc. As memories of user processes and hardware registers lay beyond the visibility of kernel's C0 variables the primitives necessarily contain inline assembly code.

The kernel layer which contains the implementation of the primitives is a $\mathrm{C0_A}$ program called the CVM framework. Besides primitives, the CVM framework contains implementation of process-context switch procedures [107], an elementary dispatcher, handlers for page faults [8, 105] as well as elementary hard-disk drivers [7, 5]. The framework takes care of an abstract kernel invocation. For this the CVM framework contains a declaration of an external call of an abstract-kernel dispatcher.

Because of external calls neither an abstract kernel nor the CVM framework can run standalone on a target hardware. In order to obtain an executable program they have to be linked together. The result of this linking is called a *concrete kernel*.

5.1.4 CVM Primitives

Primitives are basic means that CVM provides to implementors of operating-system services. Table 5.1 briefly describes 14 primitives comprised by CVM[1].

[1]The last two primitives — `cvm_get_vm_word` and `cvm_set_vm_word` — were implemented in an intermediate release of the CVM code. They have been verified in the scope of this thesis, but, however,

Table 5.1: CVM primitives

Name	Description
`cvm_reset`()	initializes the memory and the registers of a process
`cvm_clone`()	clones a process
`cvm_alloc`()	gives additional memory to a process
`cvm_free`()	releases a given amount of the memory of a process
`cvm_copy`()	copies data between processes
`cvm_get_vm_gpr`()	reads a register of a process
`cvm_set_vm_gpr`()	writes a register of a process
`cvm_virt_io`()	copies data between a device and a process
`cvm_in_word`()	reads a word from a device
`cvm_out_word`()	writes a word to a device
`cvm_setmask`()	masks external interrupts
`cvm_load_os`()	load a process image from the boot region
`cvm_get_vm_word`()	reads a word from the virtual memory of a process
`cvm_set_vm_word`()	writes a word to the virtual memory of a process

5.1.5 Computations

The state space of the CVM which is formally defined further in Section 5.2 comprises components for user processes, the abstract kernel, and devices. The transition function of the CVM model formally defined in Section 5.3 distinguishes, therefore, three top-level cases of an execution corresponding to the mentioned CVM components.

The case distinction is guided by two variables: an execution-sequence element and a current-process identifier. The execution-sequence element is a parameter of the CVM transition function. It defines the first branch of the transition function: whether the devices make a step or not. If the devices do not make a step we analyze the current-process identifier and determine whether the kernel or one of users makes progress. The current-process identifier is comprised by the CVM state. We will call these three cases of an execution, a *devices step*, a *kernel step*, and a *user step*, respectively.

The transition function of CVM has two parameters: a CVM state and an execution-sequence element. In case the execution-sequence element corresponds to some device, it comprises inputs from the external environment for this device. The step function returns an updated CVM state[2].

Devices step. A devices step boils down to an external step of the device specified by the execution-sequence element. Recall from Section 3.3 that an external step means a step taken as a response to an input generated by the external environment. The effect of a device step is to update the devices component of the CVM model.

Kernel step. If the current-process identifier component of the CVM model has a special value corresponding to the kernel process then a kernel step is taken. Kernel steps come in three flavors: (i) the kernel stays in the idle state, (ii) the kernel finishes

excluded from the CVM's latest version.

[2]In the formal theories the CVM step function additionally returns outputs to the external environment. We do not treat them in this document.

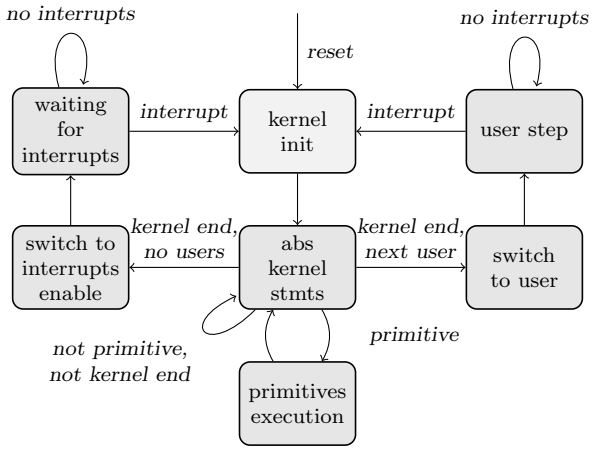

Figure 5.1: States and transitions of CVM.

its execution by switching to the idle state or to a user process, and (iii) the kernel performs a step of the abstract kernel component. The last step distinguishes between an ordinary C0 step of the abstract kernel and a primitive execution. The primitive execution occurs when the abstract kernel wants to invoke one of the CVM primitives. However, primitives are only declared in the code of the abstract kernel and called externally. Recall from Section 3.5 that the transition function of the C0 small-step semantics does not define effects of the external-call statement. Therefore a special treatment of such calls is required. For the case of a primitive invocation the CVM transition function defines which effect the primitive has on the user processes and/or the kernel.

User step. In case the current-process identifier has some value different from the one that corresponds to the kernel, the user process specified by the identifier makes a step. A user step distinguishes three cases: (i) an uninterrupted step, (ii) an interrupted step with an abort of user execution, and (iii) a step with interrupt which nevertheless allows us to perform a step of the user machine before interrupt handling. In the first case the step boils down to an update of the user process configuration by the VAMP assembly transition function. In case an interrupt occurs during the user step, the user has to be suspended and the kernel has to be invoked. The kernel invocation is required in order to handle the interrupt. The actions taken in the third case are simply a composition of the first and the second case.

The states and transitions of the CVM configuration excluding the devices are reflected in Figure 5.1.

5.2 Configurations

In this section we introduce a formal definition of the CVM model configurations and the initial configuration of CVM.

We denote the number of processes including the kernel that are allowed to run in

our system by the constant PID_MAX. We set

$$\texttt{PID\_MAX} \stackrel{\text{def}}{=} 128.$$

For identifiers of user processes we introduce the following data type

$$\mathit{Procnum} \stackrel{\text{def}}{=} \{1..\texttt{PID\_MAX} - 1\}.$$

Altogether user processes of the system are modeled as a mapping from process identifiers to VAMP assembly configurations. We use the following data type for that:

$$\mathit{Userprocs} \stackrel{\text{def}}{=} \mathit{Procnum} \mapsto C_{\text{ASM}}.$$

Configurations c_{CVM} of the Communication Virtual Machines model are represented by the the record C_{CVM} which has the following components:

- $ak :: C_{\text{C0}}^{\text{mono}}$: the configuration of the abstract kernel (including a type environment and a function table),
- $ups :: \mathit{Userprocs}$: the mapping of user processes,
- $ds :: C_{\text{DS}}$: the configuration of the devices system,
- $cup :: \mathit{Procnum}_{\perp}$: the current-process identifier, and
- $sr :: \mathbb{N}$, the status-register content used as a mask for interrupts.

The abstract kernel is modeled by the monolithic C0 small-step semantics configuration. Each user process is modeled by the VAMP assembly semantics. Configurations of the CVM model are by no means restricted to any particular instantiation of the devices system component. However, we will instantiate the devices-system component with the devices system of the underlying physical combined system from which the swap hard disk is removed. The identifier of a current process is modeled by option type $\mathit{Procnum}_{\perp}$: the value $\perp$ corresponds to the kernel while any value *pid* which belongs to the set *Procnum* corresponds to the process with number *pid*. The status register component represent the status register shared between the user processes. This design decision was taken due to the following reason.

The status register in VAMP ISA and assembly models is used to store interrupt masks. In particular, it is used to mask out interrupts from devices. Consider a scenario in which some user pid_1 masks out all devices but one. Let *did* be the identifier of this unmasked device. By this, the user pid_1 claims that it waits for an interrupt from the device *did*. The user processes are scheduled according to some policy, and eventually user pid_1 will be suspended while some other user pid_2 is resumed. During the execution of the process pid_2 the device *did* may produce an interrupt. However, it might be that the process pid_2 masks this interrupt out. If all other process do so as well, the process pid_1 will never see an interrupt from the desired device. By making the status register shared between the user processes we avoid such scenarios.

5.2.1 Abstract Kernel Program

Execution of the CVM semantics is parametrized with an abstract kernel code. It will be passed as an argument to many CVM-related definitions further in the thesis, e.g., the definition of the initial CVM configuration. We denote the abstract kernel C0 program as π_{AK}.

5.2.2 Initial Configuration

We start discussing the initial configuration of the CVM model by defining initial values of its components.

Initial Abstract Kernel

The standard approach of C0 semantics to create an initial local stack and a program rest is as follows. The local stack consists of one frame corresponding to the frame of the main function, the program rest is the body of the main function (except for the last return statement). If we choose the abstract-kernel dispatcher as the main function this does not fit our needs because we lose the last statement in the abstract kernel and cannot get the result of the abstract kernel execution. That is why, we artificially create a frame with only one variable of name that is not used neither in abstract kernel nor in CVM implementation (`abs_kernel_res`). We define the program rest as a call to `dispatcher_kernel`() with `abs_kernel_res` as a return variable.

The abstract-kernel dispatcher function has two formal parameters corresponding to the values of exceptional cause and exceptional data registers. The abstract kernel uses these values to proceed with interrupts and system calls. Below we define a function for the program rest of the abstract kernel. We will use it every time we need to obtain a kernel configuration after an interrupt signal. It is used in the definition of the initial CVM configuration as well.

Definition 5.1 (Initial program rest of the abstract kernel) Let *eca* and *edata* be natural numbers. The initial program rest of the abstract kernel is defined with the function

$$ak_{\text{prog}} :: \mathbb{N} \times \mathbb{N} \mapsto \mathit{Stmt},$$

$$\begin{aligned} ak_{\text{prog}}(\mathit{eca}, \mathit{edata}) \stackrel{\text{def}}{=}\ & \mathit{sCall}(\texttt{abs\_kernel\_res}, \\ & \texttt{dispatcher\_kernel}, \\ & [\mathit{lit}(\mathit{unsgnd}(\mathit{eca})), \mathit{lit}(\mathit{unsgnd}(\mathit{edata}))], \\ & 0). \end{aligned}$$

Isabelle: `cvm/kernel_step.abs_kernel_call_stmt`

The initial local-memory stack of the abstract kernel contains only a single frame with the variable `abs_kernel_res`. We do not care about the value of this variable as well as about the return g-variable of the frame.

Definition 5.2 (Initial local-memory stack of the abstract kernel) The initial value of the single frame in the local-memory stack of the abstract kernel is defined by the constant $ak_{\text{frame}} :: \mathit{Mframe}$:

$$\begin{aligned} ak_{\text{frame}}.\mathit{ct} &= \mathsf{A}, \\ ak_{\text{frame}}.\mathit{st} &= [(\texttt{abs\_kernel\_res}, \mathit{unsgnd}_{\text{T}})], \\ ak_{\text{frame}}.\mathit{init} &= \emptyset. \end{aligned}$$

The initial local-memory stack of the abstract kernel is defined by the constant $ak_{\text{stack}} :: \mathit{Mframe} \times \mathit{Gvar}^*$:

$$ak_{\text{stack}} \stackrel{\text{def}}{=} [(ak_{\text{frame}}, \mathsf{A})].$$

Isabelle: `cvm/kernel_step.abs_kernel_stack`

With the help of Definitions 5.1 and 5.2 we introduce the function which takes some C0 small-step semantics configuration of the abstract kernel and transforms its local-memory stack and program rest to their initial states. We call this process *start of the abstract kernel*.

Definition 5.3 (Start of the abstract kernel) Let ak be a C0 small-step semantics configuration of the abstract kernel and let eca and $edata$ be natural numbers. The function

$$\mathit{start\text{-}ak} :: C_{\mathrm{C0}}^{\mathrm{mono}} \times \mathbb{N} \times \mathbb{N} \mapsto C_{\mathrm{C0}}^{\mathrm{mono}},$$

$$\mathit{start\text{-}ak}(ak, eca, edata) \stackrel{\mathrm{def}}{=} ak'$$

yields the updated configuration ak' by coping all components of the C0 small-step semantics configuration from ak except for the local-memory stack and program rest which are set to their initial values:

$$\begin{aligned} ak'.mem.lm &= ak_{\mathrm{stack}}, \\ ak'.prog &= ak_{\mathrm{prog}}(eca, edata). \end{aligned}$$

Isabelle: `cvm/kernel_step.start_abs_kernel`

Finally, we can define the initial configuration of the abstract kernel. For this we construct a C0 small-step semantics configuration with an arbitrary program rest and local-memory stack. The global- and heap-memory frames of this configuration are set to their initial values by means of the function defined in Section 3.5.5. We start the abstract kernel from this configuration and by this obtain the initial configuration of the abstract kernel.

Definition 5.4 (Initial configuration of the abstract kernel) Let $\hat{c}$ be a monolithic C0 small-step semantics configuration, such that:

$$\begin{aligned} \hat{c}.te &= \pi_{\mathrm{AK}}.te, \\ \hat{c}.ft &= \pi_{\mathrm{AK}}.ft, \\ \hat{c}.mem.gm &= init_{\mathrm{gm}}(\pi_{\mathrm{AK}}.gst), \\ \hat{c}.mem.lm &= \mathsf{A}, \\ \hat{c}.mem.hm &= init_{\mathrm{hm}}, \\ \hat{c}.prog &= \mathsf{A}. \end{aligned}$$

The initial configuration of the abstract kernel is constructed by means of the following function:

$$ak_{\mathrm{init}} :: \Pi_{\mathrm{C0}} \mapsto C_{\mathrm{C0}}^{\mathrm{mono}},$$

$$ak_{\mathrm{init}}(\pi_{\mathrm{AK}}) \stackrel{\mathrm{def}}{=} \mathit{start\text{-}ak}(\hat{c}, 1, 0).$$

Isabelle: `cvm/cvm_correct/cvm_init.init_cvm`

Note, that the values of eca and $edata$ parameters of the function *start-ak* correspond to the reset interrupt.

Initial User Processes

In the initial state a user process has its program counters as well as all general-purpose registers set to zero values. The memory is arbitrary. Initialization of the special-purpose registers is more involved.

- The register *sr* is set to $8254 = \langle 0^{18}10000000111110 \rangle$ which masks out all interrupts except for the illegal, misalignment, page faults, trap, and timer. All of these interrupts except for the timer are unmaskable; we do not allow to mask the timer interrupt in order to guarantee liveness of the system.
- The register *pto* is set to $1024 + (pid - 1) \cdot 9$ where *pid* is a process-identifier — this distributes page-table origins across the page table space with equal segments between the origins of each two consecutive processes. To implement 4GB of virtual memory the page tables of user processes altogether should contain 2^{20} entries or 2^{10} pages of entries. Initially this provides 8 pages for each process. To align page tables of each process to page borders we add additionally one page to each process, i.e., 9.
- The register *ptl* is set to -1 which denotes that no process has allocated virtual memory.
- The register *mode* is set to 1 which correspond to the user mode.
- All other special-purpose registers are set to zero.

The initial configuration of a user processes is introduced formally in the following definition.

Definition 5.5 (Initial configuration of the user processes) The initial configuration of the user processes

$$ups_{\text{init}} :: Userprocs$$

is obtained by initializing all processes following the text above:

$$\begin{aligned}
ups_{\text{init}}(pid).dpc &= 0,\\
ups_{\text{init}}(pid).pc &= 0,\\
ups_{\text{init}}(pid).gpr &= 0^{32},\\
ups_{\text{init}}(pid).spr[r] &= \begin{cases} 8254 & \text{if } r = sr \\ 1024 + (pid-1)\cdot 9 & \text{if } r = pto \\ -1 & \text{if } r = ptl \\ 1 & \text{if } r = mode \\ 0 & \text{otherwise} \end{cases},\\
ups_{\text{init}}(pid).m &= \mathsf{A}.
\end{aligned}$$

Isabelle: `cvm/cvm_correct/cvm_init.init_process`

Initial Devices

Naturally, after a computer power up we know nothing about states of the devices. The only requirement imposed by the CVM model is that there must be no swap hard disk

because the swapping is invisible in the model. Thus, we define the initial state of the devices system by means of the function which takes some configuration of the devices system and removes the swap hard disk from it. We denote the position (index) of the hard disk by the constant `SWAP_DID` which could be instantiated later according to a particular CVM implementation.

Definition 5.6 (Initial configuration of the devices system) Let ds be a devices system. The initial configuration of the devices system in the CVM model is obtained by making the swap hard disk an illegal device:

$$ds_{\text{init}} :: C_{\text{DS}} \mapsto C_{\text{DS}},$$
$$ds_{\text{init}}(ds) \stackrel{\text{def}}{=} ds',$$

such that

$$ds'(did) = \begin{cases} \textit{idle-dev} & \text{if } did = \texttt{SWAP\_DID} \\ ds(did) & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/cvm_correct/cvm_init.init_dev`

Altogether

Exploiting the definitions of initial configurations of the individual CVM components we are able to define the initial configuration of the whole CVM. Initial configurations of the abstract kernel, user processes, and devices system are stated with the help of Definitions 5.4, 5.5, and 5.6, respectively. The current-process identifier is set to $\perp$ while the value of the status register component is set to 8254 due to the same reasons as in Definition 5.5.

Definition 5.7 (Initial CVM configuration) Let ds be a devices system. The initial configuration c^0_{CVM} is obtained by means of the function

$$cvm_{\text{init}} :: \Pi_{\text{C0}} \times C_{\text{DS}} \mapsto C_{\text{CVM}},$$
$$cvm_{\text{init}}(\pi_{\text{AK}}, ds) \stackrel{\text{def}}{=} c^0_{\text{CVM}},$$

such that

$$\begin{aligned} c^0_{\text{CVM}}.ak &= ak_{\text{init}}(\pi_{\text{AK}}), \\ c^0_{\text{CVM}}.ups &= ups_{\text{init}}, \\ c^0_{\text{CVM}}.ds &= ds_{\text{init}}(ds), \\ c^0_{\text{CVM}}.cup &= \perp, \\ c^0_{\text{CVM}}.sr &= 8254. \end{aligned}$$

Isabelle: `cvm/cvm_correct/cvm_init.init_cvm_sys`

5.3 Semantics

This section formally defines the transition function of the CVM model. We start the section by introducing interrupts to the VAMP assembly model which we use to define user processes. The transition function of the CVM model distinguishes three cases: a user step, a kernel step, and a devices step. We proceed by formalizing each of the cases.

5.3.1 Dealing with Interrupts

Recall from Section 3.2 that the VAMP assembly model lacks interrupts. The decision not to integrate an interrupt mechanism into this model is made because all assembly-written system-software we treat in the project do not produce unwanted interrupts. However we decided to model user processes by means of the VAMP assembly semantics as well. In general, we do not know what code a user wants to execute — a user program easily may produce an interrupt. Because of that we introduce interrupts into the VAMP assembly model.

Interrupt Signals

In the following we define predicates over VAMP assembly configurations c_{ASM} which denote that a particular interrupt takes place. The definitions are stated in the same way as in the formal specification of the VAMP ISA model and the respective equivalence theorems are proven. A general idea behind the definitions is that interrupt signals are ordered according to their indexes (cf. Table 3.3): an interrupt with index j is defined as an absence of all interrupts with indices $i < j$ plus an essential condition for the interrupt j.

Instruction misalignment. In Section 3.2 we have already defined the predicate $\mathit{no\text{-}imal}(c_{\mathrm{ASM}})$ which denotes that no instruction misalignment occurs in the configuration c_{ASM}. Negating this predicate gives us the definition for an instruction misalignment interrupt:

$$\mathit{is\text{-}imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} \neg \mathit{no\text{-}imal}(c_{\mathrm{ASM}}).$$

Page-table length exception on fetch. In case there is an attempt to fetch an instruction from a memory address beyond the amount of available virtual memory a page-table length exception on fetch (PTL exception on fetch, shortly) occurs. The memory address at which an instruction is supposed to be fetched is specified by the delayed program counter $c_{\mathrm{ASM}}.\mathit{dpc}$. The *ptl* register denotes the number of the last virtual memory page a user process has, e.g., 0 denotes that one user-memory page is allocated, 1 denotes two pages, etc. The value -1 denotes that a use has no virtual memory. Then, a page-table length exception on fetch is formally defined as follows:

$$\begin{aligned} \mathit{is\text{-}ptlexcp\text{-}f}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=}\ & \neg \mathit{is\text{-}imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ \wedge\ & c_{\mathrm{ASM}}.\mathit{dpc} \geq \mathit{i2n}((c_{\mathrm{ASM}}.\mathit{spr}[\mathit{ptl}] + 1) \cdot \texttt{PAGE\_SIZE}). \end{aligned}$$

Illegal instruction. An illegal-instruction interrupt occurs in a configuration c_{ASM} in three cases: (i) the current instruction $\mathit{instr} = \mathit{instr}(c_{\mathrm{ASM}})$ cannot be decoded, or (ii) the return from exception is attempted in user mode, or (iii) special move instructions are executed in user mode. Formally, for a user machine:

$$\begin{aligned} \mathit{is\text{-}ill}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=}\ & \neg \mathit{is\text{-}imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ \wedge\ & \neg \mathit{is\text{-}ptlexcp\text{-}f}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ \wedge\ & (\ \ \neg \mathit{decodable}(c_{\mathrm{ASM}}.m_{\mathit{word}}(c_{\mathrm{ASM}}.\mathit{dpc})) \\ & \quad \vee \mathit{is\text{-}instr\text{-}rfe}(\mathit{instr}) \\ & \quad \vee \mathit{is\text{-}instr\text{-}movi2s}(\mathit{instr}) \\ & \quad \vee \mathit{is\text{-}instr\text{-}movs2i}(\mathit{instr})). \end{aligned}$$

Data misalignment. In Section 3.2 we have already defined the predicate $\mathit{no\text{-}dmal}(c_{\mathrm{ASM}})$ denoting absence of data misalignments in the assembly configuration c_{ASM}. We use the negation of this predicate and the absence of higher priority interrupts in order to define a data-misalignment signal:

$$\begin{array}{lcl} \mathit{is\text{-}dmal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) & \stackrel{\mathrm{def}}{=} & \neg \mathit{is\text{-}imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{is\text{-}ptlexcp\text{-}f}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{is\text{-}ill}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{no\text{-}dmal}(c_{\mathrm{ASM}}). \end{array}$$

Page-table length exception on load/store. This exception is close in meaning to a PTL exception on fetch. Suppose a load or store operation takes place, which is expressed as $\mathit{is\text{-}ls}(\mathit{instr}(c_{\mathrm{ASM}}))$. In case a memory address of the operation, denoted as $\mathit{ls\text{-}target}(c_{\mathrm{ASM}})$, has a value beyond the amount of available virtual memory a PTL exception on load/store occurs. Formally:

$$\begin{array}{lcl} \mathit{is\text{-}ptlexcp\text{-}ls}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) & \stackrel{\mathrm{def}}{=} & \neg \mathit{is\text{-}imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{is\text{-}ptlexcp\text{-}f}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{is\text{-}ill}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{is\text{-}dmal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \mathit{is\text{-}ls}(\mathit{instr}(c_{\mathrm{ASM}})) \\ & \wedge & \mathit{ls\text{-}target}(c_{\mathrm{ASM}}) \geq \mathit{i2n}((c_{\mathrm{ASM}}.\mathit{spr}[\mathit{ptl}] + 1) \cdot \mathtt{PAGE\_SIZE}). \end{array}$$

Trap. A trap interrupt happens whenever the trap instruction is executed. Section 3.2 defines the predicate $\mathit{is\text{-}instr\text{-}trap}(\mathit{instr})$ over instruction instr which we exploit here:

$$\begin{array}{lcl} \mathit{is\text{-}trap}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) & \stackrel{\mathrm{def}}{=} & \neg \mathit{is\text{-}imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{is\text{-}ptlexcp\text{-}f}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \neg \mathit{is\text{-}ill}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ & \wedge & \mathit{is\text{-}instr\text{-}trap}(\mathit{instr}(c_{\mathrm{ASM}})). \end{array}$$

Overflow. In order to define an overflow interrupt signal let us first define the predicate $\mathit{is\text{-}arith\text{-}ovf}(c_{\mathrm{ASM}}, \mathit{instr})$ which holds in case the results of particular arithmetic operations cannot be represented as VAMP assembly integer numbers. The predicate is defined by induction on the instruction.

$$\begin{array}{lcl} \mathit{is\text{-}arith\text{-}ovf}(c_{\mathrm{ASM}}, \mathtt{addio}(\mathit{rd}, \mathit{rs}, \mathit{imm})) & \stackrel{\mathrm{def}}{=} & \neg \mathbb{Z}_{32}\surd(\mathit{gpr\text{-}read}_{\mathrm{ASM}}(c_{\mathrm{ASM}}.\mathit{gpr}, \mathit{rs}) + \mathit{imm}) \\ \mathit{is\text{-}arith\text{-}ovf}(c_{\mathrm{ASM}}, \mathtt{subio}(\mathit{rd}, \mathit{rs}, \mathit{imm})) & \stackrel{\mathrm{def}}{=} & \neg \mathbb{Z}_{32}\surd(\mathit{gpr\text{-}read}_{\mathrm{ASM}}(c_{\mathrm{ASM}}.\mathit{gpr}, \mathit{rs}) - \mathit{imm}) \\ \mathit{is\text{-}arith\text{-}ovf}(c_{\mathrm{ASM}}, \mathtt{addo}(\mathit{rd}, \mathit{rs}_1, \mathit{rs}_2)) & \stackrel{\mathrm{def}}{=} & \neg \mathbb{Z}_{32}\surd(\mathit{gpr\text{-}read}_{\mathrm{ASM}}(c_{\mathrm{ASM}}.\mathit{gpr}, \mathit{rs}_1) \\ & & \quad + \mathit{gpr\text{-}read}_{\mathrm{ASM}}(c_{\mathrm{ASM}}.\mathit{gpr}, \mathit{rs}_2)) \\ \mathit{is\text{-}arith\text{-}ovf}(c_{\mathrm{ASM}}, \mathtt{subo}(\mathit{rd}, \mathit{rs}_1, \mathit{rs}_2)) & \stackrel{\mathrm{def}}{=} & \neg \mathbb{Z}_{32}\surd(\mathit{gpr\text{-}read}_{\mathrm{ASM}}(c_{\mathrm{ASM}}.\mathit{gpr}, \mathit{rs}_1) \\ & & \quad - \mathit{gpr\text{-}read}_{\mathrm{ASM}}(c_{\mathrm{ASM}}.\mathit{gpr}, \mathit{rs}_2)) \end{array}$$

For all remaining instructions the predicate returns false.

Having this, the overflow interrupt is defined as follows:

$$\begin{aligned} \textit{is-ovf}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} & \neg \textit{is-imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ \wedge & \neg \textit{is-ptlexcp-f}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ \wedge & \neg \textit{is-ill}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ \wedge & \textit{is-arith-ovf}(c_{\mathrm{ASM}}, \textit{instr}(c_{\mathrm{ASM}})). \end{aligned}$$

Interrupt Mechanism

So far we have defined individual predicates for particular internal interrupts that may occur in an assembly configuration c_{ASM}. Our next goal is to define a JISR (jump to interrupt-service routine) signal at the assembly level.

We collect all internal interrupts into a bit vector of internal-interrupt signals. The next definition introduces the function *int-intr-bv* which is used for that. The resulting vector of internal interrupts is constructed according to interrupt indices from Table 3.3 and is represented in little-endian encoding.

Definition 5.8 (Bit vector of internal interrupts) A bit vector collecting all internal interrupt signals in a VAMP assembly configuration c_{ASM} is computed by means of the function

$$\begin{aligned} \textit{int-intr-bv}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} & [\textit{bool2bit}(\textit{is-ovf}_{\mathrm{ASM}}(c_{\mathrm{ASM}})), \\ & \textit{bool2bit}(\textit{is-trap}_{\mathrm{ASM}}(c_{\mathrm{ASM}})), \\ & \textit{bool2bit}(\textit{is-ptlexcp-ls}_{\mathrm{ASM}}(c_{\mathrm{ASM}})), \\ & \textit{bool2bit}(\textit{is-ptlexcp-f}_{\mathrm{ASM}}(c_{\mathrm{ASM}})), \\ & \textit{bool2bit}(\textit{is-imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \vee \textit{is-dmal}_{\mathrm{ASM}}(c_{\mathrm{ASM}})), \\ & \textit{bool2bit}(\textit{is-ill}_{\mathrm{ASM}}(c_{\mathrm{ASM}}))]. \end{aligned}$$

Isabelle: `cvm/cvminterrupts.int_interrupts_bv`

Among all the considered internal interrupts only an overflow interrupt is maskable. All external interrupts besides the reset are maskable. In the following we assume *msk* to be a mask pattern for both internal and external interrupts. We represent *msk* as a natural number. The next definition formally introduces the internal part of the masked cause bit vector.

Definition 5.9 (Bit vector of an internal masked cause) Let c_{ASM} be a VAMP assembly configuration, and let *msk* be an interrupt-mask pattern. A bit vector of an internal masked cause is computed by means of the function

$$mca^{int}_{Bv} :: C_{\mathrm{ASM}} \times \mathbb{N} \mapsto Bv,$$

$$mca^{int}_{Bv}(c_{\mathrm{ASM}}, msk)[i] \stackrel{\mathrm{def}}{=} \begin{cases} 0 & \text{if } i = 5 \ \wedge \ \textit{bin}(msk)[6] = 0 \\ \textit{int-intr-bv}(c_{\mathrm{ASM}})[i] & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/cvminterrupts.int_mca_bv`

In a similar fashion we define a function which computes a bit vector of an external masked cause. In Section 3.4 we have defined the function *intr-dev-bv* which collects

external interrupt signals from a devices system in a single bit vector. By applying a bitwise conjunction with an appropriate part of the mask we obtain a bit vector of an external masked cause.

Definition 5.10 (Bit vector of an external masked cause) Let c_{DS} be a devices system and let *msk* be an interrupt-mask pattern. A bit vector of an external masked cause is computed by means of the function

$$mca^{ext}_{Bv} :: C_{\mathrm{DS}} \times \mathbb{N} \mapsto Bv,$$

$$mca^{ext}_{Bv}(c_{\mathrm{DS}}, msk)[i] \stackrel{\mathrm{def}}{=} bool2bit(bin(msk)[13+i] = 1 \ \wedge \ intr\text{-}dev\text{-}bv(c_{\mathrm{DS}})[i] = 1).$$

Isabelle: `cvm/cvminterrupts.ext_mca_bv`

We have to deal with the JISR signal in two cases: (i) when an interrupt occurs during a user execution, and (ii) when the function `cvm_wait`() is being executed. In the former case we consider both internal and external interrupts while in the later case we are interested only in external interrupt signals. In order to use the same formal definitions in both cases we introduce the following trick. The remaining definitions in this subsection will have an argument p of the option type $C_{\mathrm{ASM}\perp}$. Whenever the value of p is $\lfloor c_{\mathrm{ASM}} \rfloor$ we know that p corresponds to some user process modeled by an assembly configuration c_{ASM}. Here we deal with the former case and internal interrupts will be computed in the configuration c_{ASM}. Otherwise, p has value of $\perp$ which corresponds to an execution of the `cvm_wait`() function within the kernel.

Altogether, a masked-cause bit vector is defined as follows. We simply concatenate (i) a bit vector of external interrupts which stores external interrupt signals with indices from 20 down to 13, (ii) a list of seven zeros representing interrupts with indices from 12 down to 7 which we do not use in this thesis, and (iii) internal interrupts vector concatenated with a zero value of the reset interrupt in case we are computing interrupts of some user process, or a list of seven zeros in case we are computing interrupts for the `cvm_wait`() function. Actually, 11 zeros should precede this vector (to get the corresponding bit vector of length 32), but since we will convert it to the natural number we can omit them. Formally this is expressed in the following definition.

Definition 5.11 (Bit vector of a masked cause) Let p be of type $C_{\mathrm{ASM}\perp}$, let c_{DS} be devices-system configuration, and let *msk* be an interrupt-mask pattern. A bit vector of a masked cause if computed by means of the function

$$mca_{Bv} :: C_{\mathrm{ASM}\perp} \times C_{\mathrm{DS}} \times \mathbb{N} \mapsto Bv,$$

$$mca_{Bv}(p, c_{\mathrm{DS}}, msk) \stackrel{\mathrm{def}}{=} \begin{cases} mca^{ext}_{Bv}(c_{\mathrm{DS}}, msk) \circ 0^{13} & \text{if } p = \perp \\ mca^{ext}_{Bv}(c_{\mathrm{DS}}, msk) \circ 0^{6} \circ mca^{int}_{Bv}(c_{\mathrm{ASM}}, msk) \circ 0 & \text{if } p = \lfloor c_{\mathrm{ASM}} \rfloor \end{cases}.$$

Isabelle: `cvm/cvminterrupts.mca_bv`

Additionally, we define a function which represents a bit vector of a masked cause as a natural number:

$$mca_{\mathbb{N}}(p, c_{\mathrm{DS}}, msk) \stackrel{\mathrm{def}}{=} \langle mca_{Bv}(p, c_{\mathrm{DS}}, msk) \rangle).$$

Altogether, we can introduce a definition of the JISR signal which we will use for formal specification of the CVM semantics.

Definition 5.12 (JISR for CVM) Let p be of type $C_{\mathrm{ASM}\perp}$, let c_{DS} be devices-system configuration, and let *msk* be an interrupt-mask pattern. The JISR signal is on if at list one bit of the masked-cause bit vector is on. Formally this is defined by means of the following predicate:

$$\textit{is-jisr}_{\mathrm{CVM}} :: C_{\mathrm{ASM}\perp} \times C_{\mathrm{DS}} \times \mathbb{N} \mapsto \mathbb{B},$$

$$\textit{is-jisr}_{\mathrm{CVM}}(p, c_{\mathrm{DS}}, \textit{msk}) \stackrel{\mathrm{def}}{=} \exists\, i :\ \textit{mca}_{Bv}(p, c_{\mathrm{DS}}, \textit{msk})[i] = 1.$$

Isabelle: `cvm/cvminterrupts.jisr`

In case of `cvm_wait`() we do not have any parameters for the interrupt handling. In case of user internal interrupts we use $\textit{edata}_{\mathbb{N}}$ to calculate this parameter.

Definition 5.13 (Exceptional data) The exceptional data for c_{ASM}

$$\textit{edata}_{\mathbb{N}} :: C_{\mathrm{ASM}} \mapsto \mathbb{N}$$

is defined as

- the delayed program counter in case of a page fault interrupt on fetch,
- the load/store target address in case of an interrupt on load/store, and
- the immediate constant in case of a trap instruction.

In all other cases the yielded value is 0:

$$\textit{edata}_{\mathbb{N}}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} \begin{cases} c_{\mathrm{ASM}}.\textit{dpc} & \text{if } \textit{is-imal}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \ \vee\ \textit{is-ptlexcp-f}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ \textit{ls-target}(c_{\mathrm{ASM}}) & \text{if } \neg\textit{is-ill}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \ \wedge\ \textit{is-ls}(\textit{instr}(c_{\mathrm{ASM}})) \\ \textit{i2n}(\textit{imm}(\textit{instr}(c_{\mathrm{ASM}}))) & \text{if } \textit{is-trap}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) \\ 0 & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/cvminterrupts.edata_nat`

5.3.2 Transitions

We continue by defining the transition function δ_{CVM} of the CVM model. The parameters of the transition function are (i) a CVM model configuration $c_{\mathrm{CVM}} :: C_{\mathrm{CVM}}$, and (ii) an execution-sequence element $s :: \textit{SeqEl}$. In case the sequence element corresponds to a processor step either the kernel or some user makes progress. Otherwise, δ_{CVM} boils down to a step of some device.

The CVM transition function yields either an error constant $\perp$ or an updated configuration of the CVM model. Thus the signature of the CVM transition function[3] is

$$\delta_{\mathrm{CVM}} :: C_{\mathrm{CVM}} \times \textit{SeqEl} \mapsto C_{\mathrm{CVM}\perp}.$$

[3] As mentioned before, in Isabelle the definition of the CVM step function also has device outputs to the external environment together with respective device identifiers.

Depending on the execution sequence element s and the current-process identifier $c_{\mathrm{CVM}}.cup$ the CVM transition function $\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, s)$ distinguishes three cases:

- the devices step: s corresponds to some device,
- the user step: s corresponds to the processor and $c_{\mathrm{CVM}}.cup$ corresponds to some user-process identifier, and
- the kernel step: s corresponds to the processor and $c_{\mathrm{CVM}}.cup$ corresponds to the kernel.

In the remaining part of this section we define the respective functions $step_{\mathrm{devs}}$, $step_{\mathrm{user}}$, and $step_{\mathrm{kernel}}$ for each of these cases. The overall definition of the CVM transition function is formally stated as follows:

$$\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, s) \stackrel{\mathrm{def}}{=} \begin{cases} \lfloor step_{\mathrm{user}}(c_{\mathrm{CVM}}) \rfloor & \text{if } s = \mathsf{Proc} \wedge c_{\mathrm{CVM}}.cup = \lfloor pid \rfloor \\ step_{\mathrm{kernel}}(c_{\mathrm{CVM}}) & \text{if } s = \mathsf{Proc} \wedge c_{\mathrm{CVM}}.cup = \perp \\ \lfloor step_{\mathrm{devs}}(c_{\mathrm{CVM}}, did, eifi) \rfloor & \text{if } s = \mathsf{Dev}(did, eifi) \end{cases}.$$

Before going into details of user, kernel and device steps let us formally introduce a function which performs multiple steps of the CVM model. The multiple-step function of the CVM $\delta^{\mathrm{n}}_{\mathrm{CVM}}$ is defined by induction on the step number n:

$$\delta^{\mathrm{n}}_{\mathrm{CVM}} :: \mathbb{N} \times C_{\mathrm{CVM}} \times Seq \mapsto C_{\mathrm{CVM}\perp},$$

$$\begin{aligned} \delta^{0}_{\mathrm{CVM}}(c_{\mathrm{CVM}}, seq) &\stackrel{\mathrm{def}}{=} \lfloor c_{\mathrm{CVM}} \rfloor \\ \delta^{\mathrm{n}+1}_{\mathrm{CVM}}(c_{\mathrm{CVM}}, seq) &\stackrel{\mathrm{def}}{=} \begin{cases} \perp & \text{if } \delta^{\mathrm{n}}_{\mathrm{CVM}}(c_{\mathrm{CVM}}, seq) = \perp \\ \delta_{\mathrm{CVM}}(c^{n}_{\mathrm{CVM}}, seq(n)) & \text{if } \delta^{\mathrm{n}}_{\mathrm{CVM}}(c_{\mathrm{CVM}}, seq) = \lfloor c^{n}_{\mathrm{CVM}} \rfloor \end{cases}. \end{aligned}$$

5.3.3 Devices Step

We start defining individual cases of the CVM transition function by introducing device steps as they turn out to be the easiest. A device step is taken as a response to an input from the external environment. Therefore, the device step boils down to an application of the external device step function $\delta^{\mathrm{EXT}}_{\mathrm{DS}}$ introduces in Section 3.3.

Definition 5.14 (Devices step) Let c_{CVM} be a configuration of the CVM model, let *did* be a device identifier, and let *eifi* be a generalized input from the external environment. A device step is specified by the function

$$step_{\mathrm{devs}} :: C_{\mathrm{CVM}} \times Devnum \times Eifi_{\mathrm{GD}} \mapsto C_{\mathrm{CVM}}$$

which produces an updated CVM configuration

$$step_{\mathrm{devs}}(c_{\mathrm{CVM}}, did, eifi) \stackrel{\mathrm{def}}{=} c'_{\mathrm{CVM}},$$

such that

$$c'_{\mathrm{CVM}}.ds = \delta^{\mathrm{EXT}}_{\mathrm{DS}}(c_{\mathrm{CVM}}.ds, did, eifi).$$

Isabelle: `cvm/cvmstep.cvmstep`

5.3.4 User Step

User steps are modeled by the function $step_{\text{user}}$ which, in essence, distinguishes three cases: (i) a user step without interrupts, (ii) a user step with an interrupt that aborts the user execution (illegal, misalignment or PTL exception), and (iii) a user step with an interrupt which allows us to take a step (external interrupts, trap, and overflow). User steps without interrupts boil down to an application of the VAMP assembly transition function to the user process which is making a step. Steps with interrupts results in setting the current-process identifier to the kernel value and the abstract-kernel invocation. Semantics of the user processes modification depends on the kind of interrupt. In case (ii) the user is not changed, otherwise it makes a step as in case (i). Altogether, the function $step_{\text{user}}$ updates the CVM-state components for user processes, the current-process identifier, and the abstract kernel.

Definition 5.15 (User step) Let c_{CVM} be a configuration of the CVM model. A user step is specified by the function $step_{\text{user}} :: C_{\text{CVM}} \mapsto C_{\text{CVM}}$ which yields an updated CVM configuration $c'_{\text{CVM}} = step_{\text{user}}(c_{\text{CVM}})$, such that

$$\begin{aligned} c'_{\text{CVM}}.ups &\stackrel{\text{def}}{=} \textit{update-ups}(c_{\text{CVM}}), \\ c'_{\text{CVM}}.cup &\stackrel{\text{def}}{=} \textit{update-cup}(c_{\text{CVM}}), \\ c'_{\text{CVM}}.ak &\stackrel{\text{def}}{=} \textit{update-ak}(c_{\text{CVM}}). \end{aligned}$$

Isabelle: `cvm/user_step.usercompute`

Before we formally define the functions *update-ups*, *update-cup*, and *update-ak*, let us introduce the predicate which tests whether interrupts allow a user process to make progress.

Definition 5.16 (User-step progress) Let c_{CVM} be a configuration of the CVM model. We test whether it is possible to make progress in c_{ASM} by means of the predicate

$$\begin{aligned} \textit{is-progress}(c_{\text{ASM}}) \stackrel{\text{def}}{=}\ & \neg \textit{is-ill}_{\text{ASM}}(c_{\text{ASM}}) \\ \wedge\ & \neg \textit{is-imal}_{\text{ASM}}(c_{\text{ASM}}) \ \wedge\ \neg \textit{is-dmal}_{\text{ASM}}(c_{\text{ASM}}) \\ \wedge\ & \neg \textit{is-ptlexcp-f}_{\text{ASM}}(c_{\text{ASM}}) \ \wedge\ \neg \textit{is-ptlexcp-ls}_{\text{ASM}}(c_{\text{ASM}}). \end{aligned}$$

Isabelle: `cvm/user_step.user_step_progress`

Update of User Processes

The current process modeled by an assembly configuration c_{ASM} can perform a step only in case no interrupts occur in c_{ASM} or interrupts do not abort the user execution. In case the current process is able to make a step its assembly configuration c_{ASM} has to be updated with the assembly step function δ_{ASM}. Configurations of all other user process remain the same. Such update of the user processes mapping of the CVM model is formally handled in the following definition.

Definition 5.17 (Single user step) Let *ups* be a mapping of user processes and let

cup be a process identifier. A single transition of the process identified by *cup* is made by means of the function

$$\textit{one-user-step} :: \textit{Userprocs} \times \textit{Procnum} \mapsto \textit{Userprocs}$$

which yields an updated user-processes mapping $ups' = \textit{one-user-step}(ups, cup)$, such that

$$ups'(pid) \stackrel{\text{def}}{=} \begin{cases} \delta_{\text{ASM}}(ups(cup)) & \text{if } pid = cup \\ ups(pid) & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/user_step.one_user_step`

With the help of Definitions 5.16 and 5.17 we are able to formally specify how the user-processes mapping of the CVM model is updated in case of a user step.

Definition 5.18 (Update of user processes in a user step) Let c_{CVM} be a configuration of the CVM model. Let $\lfloor cup \rfloor = c_{\text{CVM}}.cup$ be the current-process identifier of c_{CVM}. The function $\textit{update-ups} :: C_{\text{CVM}} \mapsto \textit{Userprocs}$ performs a single user step of the process identified by *cup* in case this process as able to make progress:

$$\textit{update-ups}(c_{\text{CVM}}) \stackrel{\text{def}}{=} \begin{cases} \textit{one-user-step}(c_{\text{CVM}}.ups, cup) & \text{if } \textit{is-progress}(c_{\text{CVM}}.ups(cup)) \\ c_{\text{CVM}}.ups & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/user_step.userprocesses_step`

Update of the Current-Process Identifier

We update the current-process identifier in a user step as follows. If there is an interrupt raised in the user-process configuration associated with the current-process identifier, then we assign to the current-process identifier the value $\bot$. This will invoke the kernel in the next CVM step. Otherwise, we just keep the value of the current-process identifier the same. Formally this is stated in the next definition.

Definition 5.19 (Update of the current-process identifier in a user step) Let c_{CVM} be a configuration of the CVM model. Let $\lfloor cup \rfloor = c_{\text{CVM}}.cup$ be the current-process identifier of c_{CVM}. The CVM-state component for the current-process identifier is updated in a user step by means of the function

$$\textit{update-cup} :: c_{\text{CVM}} \mapsto \textit{Procnum}_{\bot},$$

$$\textit{update-cup}(c_{\text{CVM}}) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \textit{is-jisr}_{\text{CVM}}(\lfloor c_{\text{CVM}}.ups(cup) \rfloor, c_{\text{CVM}}.ds, c_{\text{CVM}}.sr) \\ cup & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/user_step.currentp_step`

Update of the Abstract Kernel

Likewise updating the current-process identifier we make a case distinction on the JISR signal when updating the abstract-kernel component of CVM. In case an interrupt occurs in the assembly configuration of the current user process we are supposed to invoke the

abstract kernel. Therefore the abstract-kernel component of the CVM state is updated by means of the function *start-ak* (cf. Definition 5.3). In case no interrupt signal is raised we simply leave the abstract-kernel component unchanged. This is formalized in the next definition.

Definition 5.20 (Update of the abstract kernel in a user step) Let c_{CVM} be a configuration of the CVM model, let $\lfloor cup \rfloor = c_{\mathrm{CVM}}.cup$ be the current-process identifier, let $c_{\mathrm{ASM}} = c_{\mathrm{CVM}}.ups(cup)$ be a VAMP assembly configuration of the current process of c_{CVM}, let $c_{\mathrm{DS}} = c_{\mathrm{CVM}}.ds$ be a devices system of c_{CVM}, and let $msk = c_{\mathrm{CVM}}.sr$ be an interrupt mask specified by the status register of c_{CVM}. Additionally, let us agree that $mca = mca_{\mathbb{N}}(\lfloor c_{\mathrm{ASM}} \rfloor, c_{\mathrm{DS}}, msk)$ and $edata = edata_{\mathbb{N}}(c_{\mathrm{ASM}})$. The function *update-ak* :: $C_{\mathrm{CVM}} \mapsto C_{\mathrm{C0}}$ yields an updated abstract-kernel component of the CVM state:

$$\mathit{update\text{-}ak}(c_{\mathrm{CVM}}) \stackrel{\mathrm{def}}{=} \begin{cases} \mathit{start\text{-}ak}(c_{\mathrm{CVM}}.ak, mca, edata) & \text{if } \mathit{is\text{-}jisr}_{\mathrm{CVM}}(\lfloor c_{\mathrm{ASM}} \rfloor, c_{\mathrm{DS}}, msk) \\ c_{\mathrm{CVM}}.ak & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/user_step.kernel_step_user`

5.3.5 Kernel Step

There are three top-level cases of a kernel step modeled by the function $step_{\mathrm{kernel}}$:

- waiting for interrupts from devices,
- finishing the kernel execution, and
- a step of the abstract kernel.

The first case models a situation when there is not even a single user-process in the system to be resumed. In this case, the kernel has no jobs to accomplish, and hence its configuration remains the same. We say that the kernel is *waiting for interrupts* in this situation. If an interrupt occurs, the kernel will be restarted.

The second case handles a switch from the kernel execution either to an execution of the next scheduled user process, or to the idle state defined in the first case.

The last case models a step of the abstract kernel. This step boils down to two cases: a simple C0 step of the abstract kernel or an execution of some CVM primitive.

In the following we formally define three functions *wait-intr*, *end-kernel*, and *exec-ak* which correspond to the three possible cases of kernel step. With the help of these functions we introduce the top-level function for kernel steps $step_{\mathrm{kernel}}$. It simply makes a case distinction between the three functions mentioned above. It is indicated by the program rest of the abstract kernel which of the three cases happens. The kernel computation boils down to *wait-intr* if the program rest contains a single assembly statement. This situation is created artificially and is formally stated in Definition 5.26 further. An end of the kernel execution modeled by *end-kernel* occurs if the program rest contains a single empty statement. This corresponds to the state when there is no remaining statements of the kernel yet to be executed. If none of these criteria is met the function $step_{\mathrm{kernel}}$ is nothing but the execution of the abstract kernel. Formally, the kernel step is introduced in the next definition.

Definition 5.21 (Kernel step) Let c_{CVM} be a configuration of the CVM model. The

function $step_{\text{kernel}} :: C_{\text{CVM}} \mapsto C_{\text{CVM}\perp}$ specifies a kernel step of CVM:

$$step_{\text{kernel}}(c_{\text{CVM}}) \stackrel{\text{def}}{=} \begin{cases} \lfloor \textit{wait-intr}(c_{\text{CVM}}) \rfloor & \text{if } \textit{is-}\textsf{asm}(c_{\text{CVM}}.ak.prog) \\ \textit{end-kernel}(c_{\text{CVM}}) & \text{if } \textit{is-}\textsf{skip}(c_{\text{CVM}}.ak.prog) \\ \textit{exec-ak}(c_{\text{CVM}}) & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/kernel_step.kernelcompute`

Waiting for Interrupts

In case the are no user processes which can be executed after a kernel computation the kernel ends in the *waiting for interrupts* state. A step of the CVM model in this situation is either a fixpoint in case there is no pending devices interrupts, or a *start of the abstract kernel*, otherwise. The next definition formalizes the case. Note that, we determine whether any device in the CVM state c_{CVM} has raised an interrupt signal by means of the function $\textit{is-jisr}_{\text{CVM}}(\perp, c_{\text{CVM}}.ds, c_{\text{CVM}}.sr)$ (cf. Definition 5.12). Its first parameter $\perp$ reflects the fact that only external interrupts from the devices $c_{\text{CVM}}.ds$ are to be considered.

Definition 5.22 (Waiting for interrupts) Let c_{CVM} be a configuration of the CVM model. The function $\textit{wait-intr} :: C_{\text{CVM}} \mapsto C_{\text{CVM}}$ yields an updated CVM state c'_{CVM} in case the devices $c_{\text{CVM}}.ds$ produce an interrupt, or has no effect on c_{CVM}, otherwise:

$$\textit{wait-intr}(c_{\text{CVM}}) \stackrel{\text{def}}{=} \begin{cases} c'_{\text{CVM}} & \text{if } \textit{is-jisr}_{\text{CVM}}(\perp, c_{\text{CVM}}.ds, c_{\text{CVM}}.sr) \\ c_{\text{CVM}} & \text{otherwise} \end{cases}.$$

The updated CVM configuration c'_{CVM} differs from c_{CVM} only in the abstract-kernel component:

$$c'_{\text{CVM}}.ak \;=\; \textit{start-ak}(c_{\text{CVM}}.ak, \textit{mca}_{\mathbb{N}}(\perp, c_{\text{CVM}}.ds, c_{\text{CVM}}.sr), 0).$$

Isabelle: `cvm/kernel_step.waiting_for_interrupts`

Similarly, to the computation of the JISR signal the computation of the (natural representation of the) masked-cause bit vector $\textit{mca}_{\mathbb{N}}(\perp, c_{\text{CVM}}.ds, c_{\text{CVM}}.sr)$ has $\perp$ as the first argument, which takes into account only external interrupts.

Kernel End

As mentioned before, kernel executions end with resuming a computation of some user process or waiting for external interrupts. A decision which of these two cases takes place is made by examining the output of last abstract-kernel run. A scheduler implemented in the abstract kernel computes the identifier of the next scheduled process and returns this value to the CVM framework. It was stated in Definition 5.1 that the identifier of the next scheduled process returned by the abstract kernel is stored in the local variable `abs_kernel_res` of the first local memory frame. Next, we define a function which retrieves this value.

Definition 5.23 (Current-process identifier returned by the abstract kernel) Let c_{CVM} be a configuration of the CVM model. The function $\textit{ak-cup} :: C_{\text{CVM}} \mapsto \mathbb{N}$

returns the value of the current-process identifier computed by (the scheduler of) the abstract kernel:

$$\textit{ak-cup}(c_{\mathrm{CVM}}) \stackrel{\mathrm{def}}{=} \textit{m2u}(\textit{value}_{\mathrm{g}}(c_{\mathrm{CVM}}.\textit{ak}.\textit{mem}, \textit{gvar}_{\mathrm{lm}}(0, \texttt{abs\_kernel\_res}))).$$

Isabelle: `cvm/kernel_step.abs_cup_value`

Having this, we can formally introduce the two possible outcomes of a kernel end. If *ak-cup* computes some value between 0 and `PID_MAX`, which corresponds to a user process the kernel ends in switching to a user. This is formalized by the function *switch-to-user* (cf. Definition 5.25 further). In case *ak-cup* returns some other value the kernel ends in switching to wait with the help of the function *swich-to-wait* (cf. Definition 5.26 below). The next definitions formalizes a kernel end function *end-kernel*.

Definition 5.24 (Kernel end) Let c_{CVM} be a configuration of the CVM model and let $\textit{is-user} = 0 < \textit{ak-cup}(c_{\mathrm{CVM}}) < \texttt{PID\_MAX}$. The function $\textit{end-kernel} :: C_{\mathrm{CVM}} \mapsto C_{\mathrm{CVM}\bot}$ specifies the *kernel end* case of a kernel step:

$$\textit{end-kernel}(c_{\mathrm{CVM}}) \stackrel{\mathrm{def}}{=} \begin{cases} \textit{switch-to-user}(c_{\mathrm{CVM}}, \textit{ak-cup}(c_{\mathrm{CVM}})) & \text{if } \textit{is-user} \\ \lfloor \textit{swich-to-wait}(c_{\mathrm{CVM}}) \rfloor & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/kernel_step.switch_from_kernel`

We continue by defining *switch-to-user* and *swich-to-wait* formally. In CVM we use the amount of virtual memory of a process to determine whether the process is created, and therefore can be resumed. If no virtual memory is allocated to a process, then we consider that this process is not created. We introduce the predicate $\textit{has-memory} :: \textit{Userprocs} \times \mathbb{N} \mapsto \mathbb{B}$, such that $\textit{has-memory}(\textit{ups}, \textit{pid})$ tests whether the process $\textit{ups}(\textit{pid})$ has allocated memory:

$$\textit{has-memory}(\textit{ups}, \textit{pid}) \stackrel{\mathrm{def}}{=} \textit{ups}(\textit{pid}).\textit{spr}[\textit{ptl}] \geq 0.$$

The act of assigning the current-process identifier component of a CVM state c_{CVM} some new value *pid* in case the process $c_{\mathrm{CVM}}.\textit{ups}(\textit{pid})$ has memory is handled by the function *switch-to-user* introduced in the next definition.

Definition 5.25 (Switch to user) Let c_{CVM} be a configuration of the CVM model and let *pid* be a process identifier. The function $\textit{switch-to-user} :: C_{\mathrm{CVM}} \times \mathbb{N} \mapsto C_{\mathrm{CVM}\bot}$ produces a CVM output comprising an updated CVM configuration c'_{CVM} in case the user process associated with *pid* has allocated virtual memory, and an error constant $\bot$ if it does not:

$$\textit{switch-to-user}(c_{\mathrm{CVM}}, \textit{pid}) \stackrel{\mathrm{def}}{=} \begin{cases} \lfloor c'_{\mathrm{CVM}} \rfloor & \text{if } \textit{has-memory}(c_{\mathrm{CVM}}.\textit{ups}, \textit{pid}) \\ \bot & \text{otherwise} \end{cases}.$$

The updated CVM state differs from the original configuration c_{CVM} only in the updated current-process identifier component:

$$c'_{\mathrm{CVM}}.\textit{cup} \;=\; \lfloor \textit{pid} \rfloor.$$

Isabelle: `cvm/kernel_step.switch_to_user`

Finally, we define switching to the state in which the kernel is waiting for external interrupts. Recall from Definition 5.21 that the *waiting for interrupts* case is distinguished by a single assembly statement in the program rest of the abstract kernel. It turns out that according to the C0 small-step semantics the abstract kernel itself cannot come to a state with such program rest. By artificially assigning the program rest the assembly statement, we introduce a way to distinguish the desired *waiting for interrupts* case from the other cases.

Definition 5.26 (Switch to wait) Let c_{CVM} be a configuration of the CVM model. The function *swich-to-wait* :: $C_{\text{CVM}} \mapsto C_{\text{CVM}}$ yields the updated CVM state $c'_{\text{CVM}} = \textit{swich-to-wait}(c_{\text{CVM}})$ which differs from the original one only in the program rest of the abstract-kernel component — it is assigned to an empty assembly statement:

$$c'_{\text{CVM}}.\textit{ak.prog} \;=\; \mathit{asm}([], 0).$$

Isabelle: `cvm/kernel_step.switch_to_wait`

Abstract-Kernel Execution

Executions of the abstract kernel come in two flavors: normal C0 steps of the abstract kernel and a primitive execution. In order to distinguish these two cases we need a formal way to determine that the abstract kernel is going to execute one of the CVM primitives.

Table 5.1 defines the set *prims* of CVM-primitives names as they appear in the implementation of the CVM framework. We determine that a C0 configuration c_{C0} is going to execute a CVM primitive by the following predicate:

$$\begin{aligned} \textit{is-prim}(c_{\text{C0}}) \;&\stackrel{\text{def}}{=}\; \textit{is-}\mathit{esCall}(\textit{stmt}(c_{\text{C0}})) \\ &\wedge\quad \textit{called-func}(\textit{stmt}(c_{\text{C0}})) \in \textit{prims}. \end{aligned}$$

It tests whether the current statement of c_{C0} is an external call to the function of some name from the set *prims*.

The case of a CVM-primitive execution and a simple C0 step of the abstract kernel are modeled by the functions *exec-prim* and *ak-step* which appear further in this section. A case distinction between them is formally done by the function *exec-ak* formally specified in the next definition.

Definition 5.27 (Abstract-kernel execution) Let c_{CVM} be a configuration of the CVM model. An abstract-kernel execution modeled by the function *exec-ak* :: $C_{\text{CVM}} \mapsto C_{\text{CVM}\perp}$ boils down to a case distinction between a primitive execution and an ordinary C0 step of the abstract kernel:

$$\textit{exec-ak}(c_{\text{CVM}}) \;\stackrel{\text{def}}{=}\; \begin{cases} \textit{exec-prim}(c_{\text{CVM}}) & \text{if } \textit{is-prim}(c_{\text{CVM}}.\textit{ak}) \\ \textit{ak-step}(c_{\text{CVM}}) & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/kernel_step.abstract_kernel_exec`

An ordinary C0 step is handled by the function *ak-step* which applies the C0 small-step transition function to the configuration of the abstract kernel. In case this C0 computational step ends in an error state, then so does the function *ak-step* itself.

Definition 5.28 (Abstract-kernel step) A step of the abstract kernel is specified by the function *ak-step* :: $C_{\mathrm{CVM}} \mapsto C_{\mathrm{CVM}\perp}$ which produces an empty output in case the C0 computation of the abstract kernel results in error, or an output which comprises an updated CVM configuration c'_{CVM}:

$$\textit{ak-step}(c_{\mathrm{CVM}}) \stackrel{\mathrm{def}}{=} \begin{cases} \perp & \text{if } \delta^{\mathrm{mono}}_{\mathrm{C0}}(c_{\mathrm{CVM}}.ak) = \perp \\ \lfloor c'_{\mathrm{CVM}} \rfloor & \text{if } \delta^{\mathrm{mono}}_{\mathrm{C0}}(c_{\mathrm{CVM}}.ak) = \lfloor ak' \rfloor \end{cases}.$$

The updated CVM state c'_{CVM} differs from the original configuration c_{CVM} only in the abstract-kernel component:

$$c'_{\mathrm{CVM}}.ak \;=\; ak'.$$

Isabelle: `cvm/kernel_step.abstract_kernel_step`

Primitive Execution

An execution of a CVM primitive modeled by the function *exec-prim* which is to be defined further proceeds according to the following scheme. First, the name of a primitive is determined. Then, a number of common preconditions of technical nature are checked. In case the preconditions hold the parameters are evaluated and *exec-prim* makes a case distinction on the primitive name and boils down to the function which defines the semantics of the primitive which has to be executed. Next we formally define common preconditions to the CVM primitives, evaluation of parameters of the CVM primitives, and the function *exec-prim*. However, the semantics of primitives we discuss separately (cf. Section 5.4).

We start with the formulation of the common preconditions to primitives. As mentioned above they are of technical nature. Assume that the current statement of the C0 configuration of the abstract kernel is an external call to some primitive. Then the preconditions require the following:

- an entry for the left-hand variable of the current call statement must be present in the top local memory frame (because primitive execution should not change the global data structure of the abstract kernel),
- all parameter expression could be evaluated without errors, and
- all parameter expressions are initialized, i.e., the predicate *is-initialized* from Section 3.5 hold for them.

We formalize these criteria in the next definition.

Definition 5.29 (Common preconditions to primitives) Let c_{C0} be a monolithic C0 small-step semantics configuration.Common preconditions to primitives are specified by the predicate

$$\begin{aligned} \mathit{PRE}_{\mathrm{prim}}(c_{\mathrm{C0}}) \;\stackrel{\mathrm{def}}{=}\; & \textit{l-var}(\textit{stmt}(c_{\mathrm{C0}})) \in \textit{vns}(\textit{lst}_{\mathrm{top}}(c_{\mathrm{C0}}.\textit{mem})) \\ \wedge\; & \forall\, e \in \{\textit{param-list}(\textit{stmt}(c_{\mathrm{C0}}))\} : \textit{reval}(c_{\mathrm{C0}}.\textit{te}, c_{\mathrm{C0}}.\textit{mem}, e) = \lfloor \textit{ct} \rfloor \\ & \qquad\qquad \wedge\; \textit{is-initialized}(c_{\mathrm{C0}}.\textit{te}, c_{\mathrm{C0}}.\textit{mem}, e). \end{aligned}$$

Isabelle: `cvm/primitive_step.kernel_primitives_pre`

Recall from Section 3.5 that C0 expressions are evaluated by means of the function *reval* which yields a content of the type $\mathbb{N} \mapsto \mathit{Mcell}_\bot$. In case evaluation brings no error the result is the mapping $\lfloor ct \rfloor$. This mapping is needed to support complex C0 types. However, when we deal with parameters of the CVM primitive we encounter only basic C0 types. Their evaluation always occupies one memory cell, namely $ct(0)$. Therefore, it is more convenient for us to have an evaluation function with the codomain of $\mathit{Mcell}_\bot$ type. The next definition introduces the function *eval-param* which respects the described property and is used for parameter evaluation on the context of CVM.

Definition 5.30 (Parameter evaluation) Let c_{C0} be a monolithic C0 small-step semantics configuration and let e be a C0 expression. The function *eval-param* :: $C_{\mathrm{C0}}^{\mathrm{mono}} \times \mathit{Expr} \mapsto \mathit{Mcell}$ evaluates e with the C0 evaluation function and takes the very first item of the obtained content in case the evaluation produced no error:

$$\mathit{eval\text{-}param}(c_{\mathrm{C0}}, e) \stackrel{\mathrm{def}}{=} \begin{cases} \mathsf{A} & \text{if } \mathit{reval}(c_{\mathrm{C0}}.te, c_{\mathrm{C0}}.mem, e) = \bot \\ ct(0) & \text{if } \mathit{reval}(c_{\mathrm{C0}}.te, c_{\mathrm{C0}}.mem, e) = \lfloor ct \rfloor \end{cases}.$$

Isabelle: `cvm/primitive_step.eval_param`

Now we can define a function which yields the value of the i-th parameter of the current call statement. We exploit the definition of the function *param-list* which takes a C0 statements and returns the list of its parameter expressions in case the statement is a call or an external call. Applying the parameter-evaluation function defined above to the i-th element of this parameter list we obtain the computed value of the corresponding expression. The next definition formally introduces a function for that.

Definition 5.31 (Parameter of a primitive) Let c_{C0} be a monolithic C0 small-step semantics configuration and let i be a natural number. We obtain the i-th parameter of the current call statement of c_{C0} be means of the function *prim-param* :: $C_{\mathrm{C0}} \times \mathbb{N} \mapsto \mathit{Mcell}$:

$$\mathit{prim\text{-}param}(c_{\mathrm{C0}}, i) \stackrel{\mathrm{def}}{=} \mathit{eval\text{-}param}(c_{\mathrm{C0}}, (\mathit{param\text{-}list}(\mathit{stmt}(c_{\mathrm{C0}}))[i])).$$

Isabelle: `cvm/primitive_step.primitive_parameter`

Having defined the common preconditions to CVM primitives and the parameter evaluation functions, we can introduce the function *exec-prim* which models a primitive execution. This function checks for the common precondition, processes primitive parameters, and by performing a case distinction decides which of the primitives has to be executed. Effects of the individual primitives are defined as functions of the form

$$\mathit{exec}_{\mathit{prim\text{-}name}} :: C_{\mathrm{CVM}} \times \ldots \mapsto C_{\mathrm{CVM}\bot},$$

where *prim-name* is a primitive name and dots are substituted with parameter types depending on a particular primitive. We introduce the semantics of these functions in Section 5.4.

Definition 5.32 (Primitive execution) Let c_{CVM} be a configuration of the CVM model, let $pn = \mathit{called\text{-}func}(\mathit{stmt}(c_{\mathrm{CVM}}.ak))$ be the name of the primitive which is supposed to be executed, and let

$$n_i = \mathit{m2u}(\mathit{prim\text{-}param}(c_{\mathrm{CVM}}.ak, i)),$$

Table 5.2: Functions defining semantics of primitives.

Value of pn	Primitive semantics function to obtain c'_{CVM}
`cvm_reset`	$exec_{\texttt{cvm\_reset}}(c_{\mathrm{CVM}}, n_0)$
`cvm_clone`	$exec_{\texttt{cvm\_clone}}(c_{\mathrm{CVM}}, n_0, n_1)$
`cvm_alloc`	$exec_{\texttt{cvm\_alloc}}(c_{\mathrm{CVM}}, n_0, n_1)$
`cvm_free`	$exec_{\texttt{cvm\_free}}(c_{\mathrm{CVM}}, n_0, n_1)$
`cvm_copy`	$exec_{\texttt{cvm\_copy}}(c_{\mathrm{CVM}}, n_0, n_1, n_2, n_3, n_4)$
`cvm_get_vm_gpr`	$exec_{\texttt{cvm\_get\_vm\_gpr}}(c_{\mathrm{CVM}}, n_0, n_1)$
`cvm_set_vm_gpr`	$exec_{\texttt{cvm\_set\_vm\_gpr}}(c_{\mathrm{CVM}}, n_0, n_1, n_2)$
`cvm_virt_io`	$exec_{\texttt{cvm\_virt\_io}}(c_{\mathrm{CVM}}, b_0, n_1, n_2, n_3, n_4, n_5)$
`cvm_in_word`	$exec_{\texttt{cvm\_in\_word}}(c_{\mathrm{CVM}}, n_0, n_1)$
`cvm_out_word`	$exec_{\texttt{cvm\_out\_word}}(c_{\mathrm{CVM}}, n_0, n_1, n_2)$
`cvm_setmask`	$exec_{\texttt{cvm\_setmask}}(c_{\mathrm{CVM}}, n_0)$
`cvm_load_os`	$exec_{\texttt{cvm\_load\_os}}(c_{\mathrm{CVM}}, n_0, n_1)$
`cvm_get_vm_word`	$exec_{\texttt{cvm\_get\_vm\_word}}(c_{\mathrm{CVM}}, n_0, n_1)$
`cvm_set_vm_word`	$exec_{\texttt{cvm\_set\_vm\_word}}(c_{\mathrm{CVM}}, n_0, n_1, n_2)$

$$b_i = \mathit{m2b}(\mathit{prim\text{-}param}(c_{\mathrm{CVM}}.\mathit{ak}, i))$$

be its parameters represented as natural numbers and booleans, respectively. We model primitive execution by means of the function $\mathit{exec\text{-}prim} :: C_{\mathrm{CVM}} \mapsto C_{\mathrm{CVM}\perp}$, which produces a new CVM configuration in case the common preconditions to primitives are satisfied, or an error constant $\perp$, otherwise:

$$\mathit{exec\text{-}prim}(c_{\mathrm{CVM}}) \stackrel{\mathrm{def}}{=} \begin{cases} c'_{\mathrm{CVM}} & \text{if } \mathit{PRE}_{\mathrm{prim}}(c_{\mathrm{CVM}}.\mathit{ak}) \\ \perp & \text{otherwise} \end{cases}$$

The new CVM configuration is obtained as shown in table 5.2.

Isabelle: `cvm/primitive_step.execprim`

5.4 Effects of Primitives

In this section we describe semantics of CVM primitives. Before that, let us introduce several helpful definitions. Recall that the register *ptl* stores the amount of virtual memory given to a process. Moreover, the possible values of this register start from -1 which denotes that the process has no memory. We abstract from this shift by one and use the function

$$\mathit{pages\text{-}used}(c_{\mathrm{ASM}}) \stackrel{\mathrm{def}}{=} \mathit{i2n}(c_{\mathrm{ASM}}[\mathit{ptl}] + 1)$$

to obtain the number of pages used by the process. We test whether an address a belongs to the virtual memory of a process *pid* by means of the predicate

$$\mathit{is\text{-}addr\text{-}in\text{-}mem}(\mathit{ups}, \mathit{pid}, a) \stackrel{\mathrm{def}}{=} a < (\mathit{ups.spr}[\mathit{ptl}] + 1) \cdot \texttt{PAGE\_SIZE}.$$

Let m_1 and m_2 be assembly memories. We copy *len* words from the second starting at address a_2 to the first at address a_1 by means of the function

$$\mathit{mem\text{-}part\text{-}copy}(m_1, a_1, m_2, a_2, \mathit{len}) \stackrel{\mathrm{def}}{=} m',$$

which yields an update memory m' such that

$$\mathit{get\text{-}data}(m', a_1, \mathit{len}) = \mathit{get\text{-}data}(m_2, a_2, \mathit{len}).$$

At all other locations m' equals to m_1.

In the following, let *ups* and *ups*$'$ be configurations of CVM user processes before and after an execution of a primitive, respectively.

Primitive `cvm_reset`**().** A process identified by *pid* is initialized by means of the primitive `cvm_reset`(*pid*), assuming, that *pid* is a number in the range $(0, \texttt{PID\_MAX})$. The program counters of the process are set to the values $\mathit{ups}'(\mathit{pid}).(\mathit{dpc}, \mathit{pc}) = (0, 4)$, general- and special-purpose registers are cleared: $\mathit{ups}'(\mathit{pid}).\mathit{gpr}[i] = 0$ for $0 \leq i < 32$, $\mathit{ups}'(\mathit{pid}).\mathit{spr}[j] = 0$ for $0 \leq j < 32 \wedge j \notin \{\mathit{ptl}, \mathit{mode}\}$. The initial value of the register $\mathit{ups}'(\mathit{pid}).\mathit{spr}[\mathit{ptl}]$ is set to -1. The mode is set to user: $\mathit{ups}'(\mathit{pid}).\mathit{spr}[\mathit{mode}] = 1$.

Primitive `cvm_clone`**().** We create a copy, or a clone, of a process identified by $\mathit{pid}_{\text{cloner}}$ by executing the primitive `cvm_clone`$(\mathit{pid}_{\text{cloner}}, \mathit{pid}_{\text{clonee}})$. The clone has the identifier $\mathit{pid}_{\text{clonee}}$. Preconditions to the primitive comprise: (i) both $\mathit{pid}_{\text{cloner}}$ and $\mathit{pid}_{\text{clonee}}$ are numbers in the range $(0, \texttt{PID\_MAX})$, (ii) no physical memory is occupied by the process $\mathit{pid}_{\text{clonee}}$: $\mathit{pages\text{-}used}(\mathit{ups}(\mathit{pid}_{\text{clonee}})) = 0$, and (iii) there is enough virtual memory to create a copy of $\mathit{pid}_{\text{cloner}}$:

$$\sum_i \mathit{pages\text{-}used}(\mathit{ups}(i)) + \mathit{pages\text{-}used}(\mathit{ups}(\mathit{pid}_{\text{cloner}})) \leq \texttt{TVM\_MAXPAGES}.$$

Here `TVM_MAXPAGES` denotes the maximum total virtual memory size measured in pages, reserved for all processes. The semantics of the primitive defines programs counters and register files of the clonee to be a copy of the cloner: $\mathit{ups}'(\mathit{pid}_{\text{clonee}}).(\mathit{dpc}, \mathit{pc}, \mathit{gpr}, \mathit{spr}) = \mathit{ups}(\mathit{pid}_{\text{cloner}}).(\mathit{dpc}, \mathit{pc}, \mathit{gpr}, \mathit{spr})$. The memory of the clonee is obtained by copying $\mathit{pages\text{-}used}(\mathit{ups}(\mathit{pid}_{\text{cloner}}))$ first pages from the cloner:

$$\begin{aligned}\mathit{ups}'(\mathit{pid}_{\text{clonee}}).m \;=\; \mathit{mem\text{-}part\text{-}copy}(&\mathit{ups}(\mathit{pid}_{\text{clonee}}).m, 0,\\ &\mathit{ups}(\mathit{pid}_{\text{cloner}}).m, 0,\\ &\mathit{pages\text{-}used}(\mathit{ups}(\mathit{pid}_{\text{cloner}})) \cdot \texttt{PAGE\_SIZE}).\end{aligned}$$

Primitive `cvm_alloc`**().** Virtual memory of a process *pid* is extended by *pgs* pages with the primitive `cvm_alloc`(*pid*, *pgs*). The preconditions of the primitive ensure that: (i) the process identifier is valid: $0 < \mathit{pid} < \texttt{PID\_MAX}$, and (ii) we do not allocate too much memory:

$$\sum_i \mathit{pages\text{-}used}(\mathit{ups}(i)) + \mathit{pgs} \leq \texttt{TVM\_MAXPAGES}.$$

The main effect of the primitive is to copy *pgs* pages from the empty memory $\mathit{zeromem}(a) = 0$ at the end of virtual memory of the process:

$$\begin{aligned}\mathit{ups}'(\mathit{pid}).m \;=\; \mathit{mem\text{-}part\text{-}copy}(&\mathit{ups}(\mathit{pid}).m, \mathit{pages\text{-}used}(\mathit{ups}(\mathit{pid})) \cdot \texttt{PAGE\_SIZE},\\ &\mathit{zeromem}, 0,\\ &\mathit{pgs} \cdot \texttt{PAGE\_SIZE}).\end{aligned}$$

As the virtual-memory amount for a process is stored in the special register *ptl* we update it as well:

$$\mathit{ups}'(\mathit{pid}).\mathit{spr}[\mathit{ptl}] \;=\; \mathit{ups}(\mathit{pid}).\mathit{spr}[\mathit{ptl}] + \mathit{n2i}(\mathit{pgs})$$

.

Primitive `cvm_free`(). Release of *pgs* virtual-memory pages associated with a process *pid* is done by means of the primitive `cvm_free`(*pid*, *pgs*). The primitive requires *pid* to be appropriately bounded: $0 < pid <$ `PID_MAX`. Since newly allocated pages are anyway initialized with zeros we do not clear the data of the released pages, but rather only update the *ptl* register. In case a user invokes `cvm_free`(*pid*, *pgs*) with *pgs* larger than the size of the virtual memory of the process *pid* we set $ups'(pid).spr[ptl] = -1$, otherwise we subtract:

$$ups'(pid).spr[ptl] \;=\; ups(pid).spr[ptl] - n2i(pgs).$$

Primitive `cvm_copy`(). In order to copy n bytes from a process pid_{from} starting at address a_{from} to a process pid_{to} at address a_{to} we invoke the primitive `cvm_copy`($pid_{\text{from}}, pid_{\text{to}}, a_{\text{from}}, a_{\text{to}}, n$). The preconditions to the primitive in case the amount to be copied is reasonable $n > 0$, are: (i) we copy between different processes: $pid_{\text{from}} \neq pid_{\text{to}}$, (ii) we suppose to copy wordwise, therefore the addresses and amount are divisible by 4: $a_{\text{from}} \bmod 4 = 0$, $a_{\text{to}} \bmod 4 = 0$, and $n \bmod 4 = 0$, (iii) both process identifiers pid_{from} and pid_{to} lie in the interval $(0,$ `PID_MAX`$)$, and (iv) the last address we copy from/to lies within the virtual memory of a respective process: *is-addr-in-mem*$(ups, pid_{\text{from}}, a_{\text{from}} + n - 1)$ and *is-addr-in-mem*$(ups, pid_{\text{to}}, a_{\text{to}} + n - 1)$. Primitive's effects are specified as:

$$\begin{aligned} ups'(pid_{\text{to}}).m \;=\; \textit{mem-part-copy}(&ups(pid_{\text{to}}).m, a_{\text{to}},\\ &ups(pid_{\text{from}}).m, a_{\text{from}},\\ &n). \end{aligned}$$

Primitive `cvm_get_vm_gpr`(). We retrieve the content of a general-purpose register r from a process *pid* by means of the primitive `cvm_get_vm_gpr`(pid, r). The preconditions to the primitive require: (i) the process identifier validity: $0 <$ *pid* $<$ `PID_MAX`, as well as (ii) the register index to address some register in the file: $r < 32$. The semantics of the primitive is to obtain the value *i2n*$(ups(pid).gpr[r])$ which is then passed to the abstract kernel as the return value of the primitive.

Primitive `cvm_set_vm_gpr`(). Under the same preconditions as for the primitive `cvm_get_vm_gpr` we write some new value v into register r of a process *pid* by means of the primitive `cvm_set_vm_gpr`(pid, r, v). The semantics is specified as: $ups'(pid).gpr[r] = n2i(v)$.

Primitive `cvm_virt_io`(). Copy operations between devices and virtual memory are realized via the primitive `cvm_virt_io`(*req*, *did*, *prt*, *pid*, a, n). It copies n bytes of data starting at address a from/to port *prt* of a devices with number *did*. The direction of the operation is given by the boolean parameter *req*: in case *req* $=$ `T` we copy from the device to the virtual memory, otherwise, in the opposite direction. The preconditions to the primitive require validity of the process and devices identifiers as well as of the port:

$$0 < pid < \texttt{PID\_MAX} \qquad 13 \leq did < 13 + 8 \qquad prt < 2^{10}.$$

The semantics of the primitive performs a case distinction on *req*. In both cases the device *did* makes series of internal step and produces an updated devices configuration ds' and list of outputs to the memory interface *mifos*. In case *req* $=$ `F`, i.e., we write to the device, the memory interface inputs for the devices steps are created by reading the virtual memory of the process *pid*: $ups(pid).m_{word}(a + i)$ for $i < n$. In case

$req = \mathsf{T}$, the memory of the process *pid* is updated with the corresponding output: $\mathit{mem\text{-}update}_{\mathrm{ASM}}(\mathit{ups}(\mathit{pid}).m, a, \mathit{mifos}[i])$ for $i < n$. Finally, in both cases the devices component of CVM is updated with the configuration ds'.

Primitive `cvm_in_word`**().** A word is read from a port *prt* of a device with number *did* by means of the primitive `cvm_in_word`(*did*, *prt*). In case parameters of the primitive are valid, an internal step of the device is attempted which delivers an updated devices configuration ds' and memory-interface output *mifo*.The desired memory word *mifo* is passed to the abstract kernel. The devices component of CVM is updated with the configuration ds'.

Primitive `cvm_out_word`**().** A word v is written to a port *prt* of a device with number *did* by means of the primitive `cvm_out_word`(*did*, *prt*, v). The preconditions to the primitive ensure that *did* and *prt* have appropriate sizes. A memory interface input for the device created from v is passed to the device *did* which triggers a device's internal step. The step results in an updated devices configuration ds' and a memory-interface output *mifo*. The devices component of CVM is updated with the configuration ds'.

Primitive `cvm_setmask`**().** A mask for external interrupts *msk* is set by means of the primitive `cvm_setmask`(*msk*). The preconditions to the primitive ensure that: (i) *msk* fits into 32 bits: $|\mathit{bin}(\mathit{msk})| < 32$, and (ii) *msk* masks out only external interrupts: $\mathit{msk} \bmod 2^{13} = 0$. The semantics of the primitive simply updates the status register of the CVM model with the value *msk*.

Primitive `cvm_load_os`**().** An abstract kernel loads an OS image from the boot region located at the beginning of the swap hard disk to (the memory of) the process *pid* by invoking the primitive `cvm_load_os`(*pgs*, *pid*). The size of the image in pages is denoted by *pgs*. The preconditions are: (i) the process identifier is valid: $0 < \mathit{pid} <$ `PID_MAX`, (ii) the last address we copy to lies within the virtual memory of the process: $\mathit{is\text{-}addr\text{-}in\text{-}mem}(\mathit{ups}, \mathit{pid}, \mathit{pgs} \cdot \texttt{PAGE\_SIZE} - 1)$, and (iii) there is enough place on the hard disk to store such image: $\mathit{pgs} \cdot \texttt{PAGE\_SIZE} \leq |\mathit{the\text{-}hd}(\mathit{ds}(\texttt{SWAP\_DID})).\mathit{sm}|$. The semantics of the primitive updates the memory of the user $\mathit{ups}(\mathit{pid}).m$ with the values read from the hard disk.

Primitive `cvm_get_vm_word`**().** We obtain a word from the virtual memory of a process *pid* at address a by means of the primitive `cvm_get_vm_word`(*pid*, a). The primitive's preconditions ensure that: (i) the address is divisible by 4: $a \bmod 4 = 0$, (ii) the process identifier *pid* is in the range $(0,$ `PID_MAX`$)$, and (iii) a addresses inside the virtual memory of the process: $\mathit{is\text{-}addr\text{-}in\text{-}mem}(\mathit{ups}, \mathit{pid}, a)$. The primitive obtains the desired memory word $\mathit{i2n}(\mathit{ups}(\mathit{pid}).m_{\mathit{word}}(a))$ and passes it then to the abstract kernel.

Primitive `cvm_set_vm_word`() Under the same preconditions as for the primitive `cvm_get_vm_word` we write a word v into the virtual memory of a process *pid* at address a by invoking the primitive `cvm_set_vm_word`(*pid*, a, v). The semantics of the corresponding memory update is:

$$\mathit{ups}'(\mathit{pid}).m \;=\; \mathit{mem\text{-}update}_{\mathrm{ASM}}(\mathit{ups}(\mathit{pid}).m, a, \mathit{n2i}(v)).$$

All primitives update the return variable of the abstract kernel's external call with the corresponding result of primitive execution if it exists, otherwise simply with zero.

The graphical sketch of the transition function $\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, s)$ could be found in Figure 5.2.

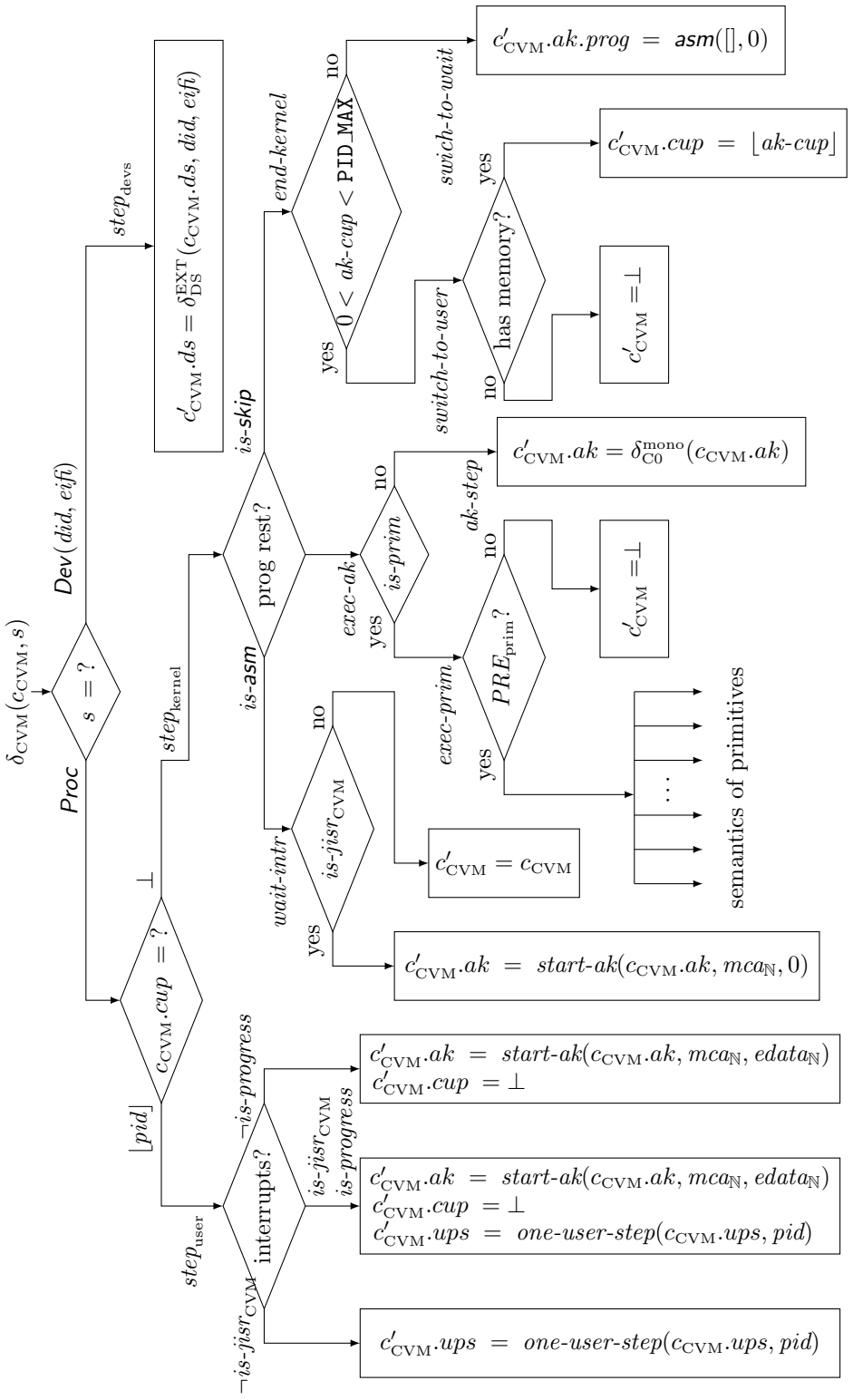

Figure 5.2: Scheme of the CVM transition function.

Chapter 6

Concrete Kernel

Contents

As mentioned before a concrete kernel, i.e., a complete kernel program that can run on a computer, is obtained by linking an abstract kernel with the CVM framework. In this chapter we formally define such linking operator. With the help of it we formally define the concrete kernel with which we will work in this thesis.

6.1 CVM Framework Implementation

Before discussing issues on linking let us consider the CVM framework implementation which will be linked with an abstract kernel.

6.1.1 The CVM Framework Structure

The CVM framework is implemented as a $C0_A$ program with approximately 1600 lines of code from which 17% constitute assembly code. The framework contains implementation of:

- process-context switch procedures saving and restoring contexts of user processes,
- a page-fault handler with its initialization code as well as elementary device drivers,
- an elementary dispatcher which decides whether an invocation of a page-fault handler is needed and calls the dispatcher of an abstract kernel, and
- 14 primitives for different operations for user processes.

Context switch. The CVM framework features procedures `init_`() and `cvm_start`() for saving and restoring contexts of user processes, respectively. The function `init_`() distinguishes a reset and non-reset cases. The former occurs after the power was switched on on the underlying processor. In this case no saving of any process contexts is required — only the kernel memory structure is created. In a non-reset case the procedure saves by means of inline assembly code the content of hardware registers into a special kernel data structure and invokes the elementary dispatcher of the CVM framework. The `cvm_start`() procedure is basically an inverse of the context save in a non-reset case. Context-switch procedures are almost fully implemented in inline assembly. The implementation is presented Section 9.2.

Page-fault handler. The page-fault handler of CVM is implemented inside the procedure `pfh_touch_addr`(). This procedure features all operations on handling page faults, software address translation, and guaranteeing for a certain page to reside in the main memory for a specified period. The page-fault handler introduces to the CVM framework a number of data structures for supporting its page-replacement strategy. Default values are written to these data structures in the page-fault handler initialization code `pfh_init`(). Both `pfh_touch_addr`() and `pfh_init`() functions are implemented in C0 without inline assembly. The needed assembly code for talking to a hard disk is isolated in elementary hard-disk drivers `write_to_disk`() and `read_from_disk`(). For details on the page-fault handler and device drivers implementation consult theses of Starostin [105] and Alkassar [5], respectively. However, we give some more details on the page-fault handler and its data structures model later in this Section.

Elementary dispatcher. The function `dispatcher`() is an elementary dispatcher of the CVM framework. In case the elementary dispatcher is called for the first time after the computer powers up the function `pfh_init`() is invoked to set up the page-fault handler data structures appropriately. Otherwise, it handles possible page faults by invoking `pfh_touch_addr`() and calls a dispatcher of the abstract kernel. This dispatcher returns to the CVM framework an identifier of the next-scheduled process or a special value in case there is no active processes. In the former case the elementary dispatcher starts the scheduled process by means of `cvm_start`(). In the latter case a special function `cvm_wait`() is called which implements the kernel idle state. We give details on the elementary dispatcher in Section 9.2.

Primitives. Microkernel primitives necessarily contain inline assembly code as they access hardware registers and memory parts which are not visible through C variables. Formal verification of mixed C and assembly code is involved because it requires reasoning in semantics of two different languages. It is, therefore, desirable to isolate inline assembly code in the minimal number of primitives. The remaining primitives then will operate only on the kernel C structures and possibly invoke those primitives that are with inline assembly. Our primitives library (Table 5.1) follows this idea: only 6 out of 14 primitives contain assembly portions. These primitives are: `cvm_copy`(), `cvm_get_vm_word`(), `cvm_set_vm_word`(), `cvm_virt_io`(), `cvm_in_word`(), `cvm_out_word`(). The three latter access devices. The three former primitives are formally verified in the scope of this thesis. We elaborate on them in Section 9.3.

6.1.2 Program of the CVM Framework

Let the program of the CVM framework be

$$\pi_{\mathrm{CVM}} \stackrel{\mathrm{def}}{=} (te_{\mathrm{CVM}}, ft_{\mathrm{CVM}}, gst_{\mathrm{CVM}}).$$

In the following, we describe which entries are contained in individual components of π_{CVM}.

Type environment. As type environments are intended to store user defined types as well as types to which a pointer in a program is declared the type environment te_{CVM} contains the following entries:

- (`int`, $\mathit{int}_{\mathrm{T}}$) for the integer type,
- (`ptspace_t`, $\mathit{arr}_{\mathrm{T}}(1152, \mathit{arr}_{\mathrm{T}}(1024, \mathit{unsgnd}_{\mathrm{T}}))$) for the page-table space array; as we already said we add one page per process to the page table to keap process page tables aligned ($1024 + 128 = 1152$), and
- an entry for a page descriptor type used by the page-fault handler.

Function table. The function table ft_{CVM} of the CVM framework implementation contains entries for:

- the CVM dispatcher (`dispatcher`()), context save and restore (`init_`(), `cvm_start`()), function for waiting of external interrupts (`cvm_wait`()),
- 14 CVM primitives,
- the declaration of the abstract-kernel dispatcher (`dispatcher_kernel`()), whose body is not defined,
- the page-fault handler (`pfh_touch_addr`()), its initialization code (`pfh_init`()), hard-disk drivers (`write_to_disk`(), `read_from_disk`()) as well as other subroutines,
- the doubly-linked list library used for the implementation of page-management algorithms.

Global symbol table. The global symbol table gst_{CVM} contains the following entries:

- the status-register variable (`SR`, $\mathit{unsgnd}_{\mathrm{T}}$),
- the current process identifier (`cup`, $\mathit{unsgnd}_{\mathrm{T}}$),
- the size of the kernel's heap (`kheap`, $\mathit{int}_{\mathrm{T}}$),
- the array of process control blocks (`pcb`, $\mathit{arr}_{\mathrm{T}}(128, \mathit{pcb\_t})$), where *pcb_t* is the type of an individual process control block; it is a structure which collects registers of a particular process,
- the page-table space array (`ptspace`, $\mathit{ptr}_{\mathrm{T}}$(`ptspace_t`)), and
- entries for variables used only in the page-fault handler.

Page-fault handler. As it was mentioned before, the page-fault handler is a significant part of CVM. It consists of two functions: `pfh_init`() for initialization of page-fault handler data structures, and `pfh_touch_addr`() which is used in two situations: (i) it is invoked on page-fault exceptions, and (ii) used by the kernel while executing CVM primitives which access user memory. In the second case the function simulates address translation for CVM which runs untranslated, and makes sure that the corresponding memory page is swapped in.

The page-fault handler implementation maintains several global data structures to manage the physical and the swap memories. The most important variables for us are: (i) the pointer to the *active* list ($\texttt{activelist}, \mathit{ptr}_{\mathrm{T}}(\texttt{pd})$), and (ii) the pointer to the *free* list ($\texttt{freelist}, \mathit{ptr}_{\mathrm{T}}(\texttt{pd})$). These lists are used to manage allocated and free user memory pages, respectively. A single element of such a list is a page descriptor `pd`. A page descriptor is a structure consisting of (i) two pointer fields (`pfh_next` and `pfh_prev`) to build a doubly linked list, and (ii) three information fields: the process identifier `pid` associated with a physical page, the index of the virtual page `vpx` corresponding to the physical page, and the index of the user page `ppx` in the physical memory. Also note, that being a part of the CVM implementation the page-fault handler accesses the process control blocks.

On a page fault the handler behaves as follows. The free list is examined in order to find out whether any unused page resides in the physical memory and could be given to a page-faulting process. If not, a page from the active list is evicted. The selected page is then filled with data loaded from the swap disk. The page table entry of the evicted page is invalidated while the valid bit of the loaded page is set.

The page-fault handler features a *copy-on-write* mechanism. When a memory page is allocated for a user process it must be filled with zeros. In order to avoid heavy swapping and zero-copying at a particular page index, we optimize the allocation process by making all freshly allocated pages point to the zero-filled page which resides at page address `ZFP`. This page is always protected. Whenever one reads from such page a zero content is provided. At a write attempt a zero-protection page-fault is signaled.

The functional correctness of the page-fault handler is shown by Starostin in [105]. However, additional peculiarities arise during an application of its correctness theorem. Since a single user instruction can cause up to two page-faults or some primitives force two pages to reside in the physical memory, we need to guarantee that the page swapped in during the previous call to the handler will not be swapped out during the current call. This liveness property could be shown only in the context of a double application of the handler's correctness theorem. Examples follow in Section 9.2.5, Section 9.3, and Chapter 10. To refer to the page-fault handler data structures we use the abstract configuration $c_{\mathrm{PFH}} :: C_{\mathrm{PFH}}$ which is abstracted from the implementation variables. Configurations of the page-fault handler are modeled by the record C_{PFH} which, among others, has the following two components:

- $c_{\mathrm{PFH}}.\mathit{active} :: \mathit{PD}^*$, the active lists, and
- $c_{\mathrm{PFH}}.\mathit{free} :: \mathit{PD}^*$, the free list.

The elements of both lists are page descriptors. A page descriptor $\mathit{pd} :: \mathit{PD}$ is a record describing a single memory page. It has three fields corresponding to the implementation structure:

- $\mathit{pd.pid} :: \mathbb{N}$, the process identifier,
- $\mathit{pd.vpx} :: \mathbb{N}$, the virtual page index, and

- $pd.ppx :: \mathbb{N}$, the physical page index.

Besides that, page-fault handler configurations maintain an abstraction of particular PCB fields, most notably the page table length of processes: $c_{\mathrm{PFH}}.pcb[pid].ptl$.

Heap symbol table. The heap symbol table is not a part of the initial program. Nevertheless, we can define it as a constant, since from the code we know, that only during the kernel initialization some elements are allocated on the heap. We use hst_{CVM} to denote the list of the elements allocated by the CVM code. Analyzing the code we can see that hst_{CVM} consists of the page table array and 1802 page descriptors needed for the page-fault handler. The elements themselves are not of our interest. We only need the information about their size: $|hst_{\mathrm{CVM}}| = \texttt{CVM\_HST\_LEN}$, and the memory allocated for them: $asize_{\mathrm{heap}}(hst_{\mathrm{CVM}})$.

Constants for the memory layout. For the ministack Theorem 4.11 application we need to assign values to some constants which define our memory layout. We do it as follows:

$$\begin{aligned}
\texttt{PROGBASE} &= 1 \cdot 2^{12},\\
\texttt{ABASE}_{\mathrm{gm}} &= 384 \cdot 2^{12},\\
\texttt{ABASE}_{\mathrm{lm}} &= 448 \cdot 2^{12},\\
\texttt{ABASE}_{\mathrm{hm}} &= 1024 \cdot 2^{12},\\
\texttt{ASIZE}_{\mathrm{hm}}^{\max} &= 5365 \cdot 2^{12},\\
\texttt{PROGEND} &= \texttt{ABASE}_{\mathrm{hm}} + \texttt{ASIZE}_{\mathrm{hm}}^{\max} = 6389 \cdot 2^{12}.
\end{aligned}$$

6.2 Linker

We formally define a linking operator $link_\pi$ which works on the source code level. It takes two $\mathrm{C0_A}$ programs and produces the third one:

$$link_\pi :: \Pi_{\mathrm{C0}} \times \Pi_{\mathrm{C0}} \mapsto \Pi_{\mathrm{C0}}.$$

Certainly, the argument programs may be simple C0 programs without assembly statements.

A $\mathrm{C0_A}$ program is defined by its type name environment, function table, and global symbol table. Therefore, next we define functions for linking each of these components. These functions will help us to define $link_\pi$ formally.

6.2.1 Linking Type Environments

The desired result of linking two type environments is a type environment which contains all types — precisely, speaking pairs containing a type name and a type — which constitute both original type environments. As some entries of a type environment might be duplicated in the environments subject to linking we have to consider such entries only once, i.e., filter second occurrences of such entries out. Formally this is stated in the next definition. Note that this definition assumes that different types in both type environments have different names.

Definition 6.1 (Linking of type environments) Linking of two type environments

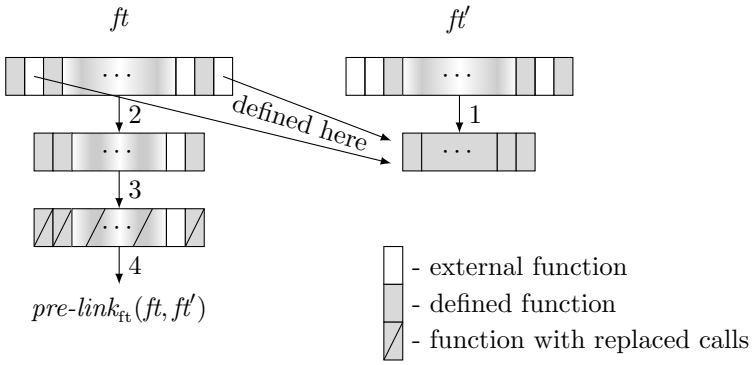

Figure 6.1: Preparing a function table for linking.

te and te' is done by means of the function

$$link_{\text{te}} :: Tenv \times Tenv \mapsto Tenv,$$

$$link_{\text{te}}(te, te') \stackrel{\text{def}}{=} te \circ [x_{te'} : x \notin te].$$

Isabelle: `cvm/linker/linker.link_tt`

6.2.2 Linking Function Tables

In order to define a function $link_{\text{ft}}$ for linking two function tables we define first an auxiliary operator $pre\text{-}link_{\text{ft}}$. It takes two function tables ft and ft' as parameters and implements the following algorithm:

1. remove all external functions from ft',
2. delete entries of external functions in ft which are defined in ft',
3. replace all external function call statements in ft by ordinary calls in case a callee is defined in ft', and
4. return the modified ft.

We illustrate this algorithm with Figure 6.1 and proceed by defining $pre\text{-}link_{\text{ft}}$ formally.

Auxiliary Linking Operator

By convention all external functions, i.e., the functions which are only declared, have their bodies as a single empty statement. For a function $f :: Func$ we define the predicate

$$is\text{-}ext\text{-}func(f) \stackrel{\text{def}}{=} is\text{-}\mathsf{skip}(f.body)$$

which holds if the function is external. The function

$$rem\text{-}ext\text{-}func(ft) \stackrel{\text{def}}{=} [x_{ft} : \neg is\text{-}ext\text{-}func(x.fd)],$$

removes all external functions from a function table *ft*. This formalizes the first step of the algorithm above.

For a function name *fn* and a function table *ft* the predicate

$$\textit{def-in-ft}(\textit{fn}, \textit{ft}) \stackrel{\text{def}}{=} \exists\, i :\ \textit{ft}[i].\textit{fn} = \textit{fn}$$

denotes that an entry for the function of name *fn* occurs in the function table *ft*. In order to specify an operation which deletes from one function table all entries of the external functions which are defined in another function table — therefore formalize the second step of the algorithm for $\textit{link}_{\text{ft}}$ — we introduce the following definition.

Definition 6.2 (Removing defined functions) Let *ft* and *ft′* be function tables. Entries of external functions in *ft* which are defined in *ft′* are removed by means of the function

$$\textit{rem-def-func} :: \textit{Functable} \times \textit{Functable} \mapsto \textit{Functable},$$

$$\textit{rem-def-func}(\textit{ft}, \textit{ft}') \stackrel{\text{def}}{=} [x_{\textit{ft}} :\ \neg(\textit{is-ext-func}(x.\textit{fd}) \wedge \textit{def-in-ft}(x.\textit{fn}, \textit{ft}'))].$$

Isabelle: `cvm/linker/linker.delete_defined_proc`

The third step of the algorithm, a replacement in one function table of external call statements to functions which are defined in another function table is handled in Definition 6.3.

Definition 6.3 (Update of external calls) Let *ft* be a function table and *s* be a statement. The function

$$\textit{ext-upd}_{\text{stmt}} :: \textit{Functable} \times \textit{Stmt} \mapsto \textit{Stmt}$$

returns an updated statement $s' = \textit{ext-upd}_{\text{stmt}}(\textit{ft}, s)$, such that

- for $s = \textit{sESCall}(e, \textit{fn}, \textit{param}, \textit{sid})$ the updated statement is obtained by replacing the external call statement by the corresponding normal call in case the function of name *fn* is defined in the function table *ft*:

$$s' \ = \ \begin{cases} \mathit{sCall}(e, \textit{fn}, \textit{param}, \textit{sid}) & \text{if } \textit{def-in-ft}(\textit{fn}, \textit{ft}) \\ \mathit{esCall}(e, \textit{fn}, \textit{param}, \textit{sid}) & \text{otherwise} \end{cases},$$

- for *s* containing sub-statements, i.e., if *s* is a loop, conditional, or composition statement, the updated statement *s′* is obtained by applying $\textit{ext-upd}_{\text{stmt}}$ to all sub-statements recursively, and
- for all other statements $s' = s$.

Let *f* be a function. The operation

$$\textit{ext-upd}_{\text{fun}} :: \textit{Functable} \times \textit{Func} \mapsto \textit{Func}$$

yields an updated function $f' = \textit{ext-upd}_{\text{fun}}(\textit{ft}, f)$ which is obtained by updating all statements of functions' *f* body with $\textit{ext-upd}_{\text{stmt}}$:

$$f'.\textit{body} \ = \ \textit{ext-upd}_{\text{stmt}}(\textit{ft}, f.\textit{body}).$$

Let ft' be a function table. A replacement of all external function call statements in ft by ordinary calls in case a callee is defined in ft' is done by means of the function

$$\textit{ext-upd}_{\text{ft}} :: \textit{Functable} \times \textit{Functable} \mapsto \textit{Functable},$$

$$\textit{ext-upd}_{\text{ft}}(ft, ft') \stackrel{\text{def}}{=} ft'',$$
$$ft''[i] \;=\; (ft'[i].fn, \textit{ext-upd}_{\text{fun}}(ft, ft'[i].fd)).$$

Isabelle: `cvm/linker/linker.ESCalls_update_{stmt,proc,pt}`

We define an alias for the functional composition of Definitions 6.2 and 6.3:

$$\textit{rem-and-upd}(ft, ft') \stackrel{\text{def}}{=} \textit{ext-upd}_{\text{ft}}(\textit{rem-def-func}(ft, ft'), ft').$$

Having this, we are able to combine all algorithm steps in a single formal definition of $\textit{link}_{\text{ft}}$.

Definition 6.4 (Auxiliary linking operator) Let ft and ft' be function tables. We define an auxiliary linking operator

$$\textit{pre-link}_{\text{ft}} :: \textit{Functable} \times \textit{Functable} \mapsto \textit{Functable},$$

$$\textit{pre-link}_{\text{ft}}(ft, ft') \stackrel{\text{def}}{=} \textit{rem-and-upd}(ft, \textit{rem-ext-func}(ft')).$$

Isabelle: `cvm/linker/linker.link_pt_with`

As all external functions that are declared in the CVM framework have their implementation in the abstract kernel and vice versa the algorithm of linking their function tables looks as follows:

1. run $\textit{pre-link}_{\text{ft}}$ with the CVM framework's function table as the first argument and the abstract kernel's function table as the second,
2. run $\textit{pre-link}_{\text{ft}}$ with the abstract kernel's function table as the first argument and the CVM framework's function table as the second, and
3. concatenate results obtained in two above steps.

Renumbering

In the third step one peculiarity has to be considered. We assume that statements are numbered consecutively starting from one in both function tables. The C0 small-step semantics requires that each statement in a program is uniquely tagged with a statement identifier. Note that the $\textit{pre-link}_{\text{ft}}$ function preserves statement identifiers. Thus, if we proceed with concatenation as described in the third step we obtain a function table which violates uniqueness of statements identifiers property. As a solution we introduce two renumbering functions $\textit{renum}^{\text{even}}_{\text{ft}}$ and $\textit{renum}^{\text{odd}}_{\text{ft}}$. Both function receive a function table as an argument and yield a modified one. The first function doubles each statement identifier in a function table while the second function multiples each statement identifier by two and additionally adds one to it. By applying respective function to the left and right sides of the concatenation in the third step of the algorithm we obtain a linked function table with unique statement identifiers. Definition 6.5 below introduces the function $\textit{renum}^{\text{even}}_{\text{ft}}$ formally.

Definition 6.5 (Even renumbering) The function

$$renum^{\text{even}}_{\text{stmt}} :: Stmt \mapsto Stmt$$

specifies the even renumbering of statements:

$$\begin{array}{lcl}
renum^{\text{even}}_{\text{stmt}}(\mathit{skip}) & \stackrel{\text{def}}{=} & \mathit{skip}, \\
renum^{\text{even}}_{\text{stmt}}(\mathit{comp}(s, s')) & \stackrel{\text{def}}{=} & \mathit{comp}(renum^{\text{even}}_{\text{stmt}}(s), renum^{\text{even}}_{\text{stmt}}(s')), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{ass}(e, e', sid)) & \stackrel{\text{def}}{=} & \mathit{ass}(e, e', 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{pAlloc}(e, t, sid)) & \stackrel{\text{def}}{=} & \mathit{pAlloc}(e, t, 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{sCall}(e, fn, es, sid)) & \stackrel{\text{def}}{=} & \mathit{sCall}(e, fn, es, 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{return}(e, sid)) & \stackrel{\text{def}}{=} & \mathit{return}(e, 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{ifte}(e, s, s', sid)) & \stackrel{\text{def}}{=} & \mathit{ifte}(e, renum^{\text{even}}_{\text{stmt}}(s), renum^{\text{even}}_{\text{stmt}}(s'), 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{loop}(e, s, sid)) & \stackrel{\text{def}}{=} & \mathit{loop}(e, renum^{\text{even}}_{\text{stmt}}(s), 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{esCall}(e, fn, es, sid)) & \stackrel{\text{def}}{=} & \mathit{esCall}(e, fn, es, 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{asm}(il, sid)) & \stackrel{\text{def}}{=} & \mathit{asm}(il, 2 \cdot sid), \\
renum^{\text{even}}_{\text{stmt}}(\mathit{xCall}(fn, es, es', sid)) & \stackrel{\text{def}}{=} & \mathit{xCall}(fn, es, es', 2 \cdot sid).
\end{array}$$

Let f be a function. The operation

$$renum^{\text{even}}_{\text{fun}} :: Func \mapsto Func$$

yields an updated function $f' = renum^{\text{even}}_{\text{fun}}(f)$ which is obtained by even renumbering of all statements in functions' f body:

$$f'.body \;=\; renum^{\text{even}}_{\text{stmt}}(f.body).$$

Let *ft* be a function table. The function

$$renum^{\text{even}}_{\text{ft}} :: Functable \mapsto Functable$$

performs even renumbering of all statements in all functions comprised by the function table *ft*:

$$renum^{\text{even}}_{\text{ft}}(ft) \stackrel{\text{def}}{=} ft'$$

$$ft'[i] \;=\; (ft[i].fn, renum^{\text{even}}_{\text{fun}}(ft[i].fd)).$$

Isabelle: `cvm/linker/linker.renumber_even_{stmt,proc,pt}`

The function $renum^{\text{odd}}_{\text{ft}}$ is defined completely analogously with the help of the functions $renum^{\text{odd}}_{\text{fun}}$ and $renum^{\text{odd}}_{\text{stmt}}$, where the last function doubles and increases by one each statement identifier *sid*, i.e., $2 \cdot sid + 1$.

Altogether

Now we are able to define the function $link_{\text{ft}}$ which links two function tables.

Definition 6.6 (Linking function tables) Linking of two function tables ft and ft' is done by means of the function

$$link_{\text{ft}} :: Functable \times Functable \mapsto Functable,$$

$$link_{\text{ft}}(ft, ft') \stackrel{\text{def}}{=} renum^{\text{even}}_{\text{ft}}(pre\text{-}link_{\text{ft}}(ft, ft')) \circ renum^{\text{odd}}_{\text{ft}}(pre\text{-}link_{\text{ft}}(ft', ft)).$$

Isabelle: `cvm/linker/linker.link_pt`

6.2.3 Linking Symbol Tables

An operator for linking symbol tables is very similar to the one for linking type environments. We concatenate the first symbol table with the part of the second symbol table from which all entries that occur in the first symbol table are removed. Formally this is expressed in the following definition.

Definition 6.7 (Linking of symbol tables) Linking of two symbol tables st and st' is done by means of the function

$$link_{\text{st}} :: Symtable \times Symtable \mapsto Symtable,$$

$$link_{\text{st}}(st, st') \stackrel{\text{def}}{=} st \circ [x_{st'} : x \notin st]$$

Isabelle: `cvm/linker/linker.link_st`

6.2.4 Linking Programs

We have formally introduced the functions for linking program's individual components in Definitions 6.1, 6.6, and 6.7. The following definition uses them for stating the linking operator over programs.

Definition 6.8 (Linking C0 programs) Linking of two C0 programs π and π' is done by means of the function

$$link_{\pi}(\pi, \pi') \stackrel{\text{def}}{=} \pi'',$$

which uses the above defined functions to link the program components:

$$\begin{aligned} \pi''.te &= link_{\text{te}}(\pi.te, \pi'.te), \\ \pi''.ft &= link_{\text{ft}}(\pi.ft, \pi'.ft), \\ \pi''.gst &= link_{\text{st}}(\pi.gst, \pi'.gst). \end{aligned}$$

Isabelle: `cvm/linker/linker.link_prog`

6.2.5 Correctness

There are two requirements to a correct linker: the produced C0 program must be

- valid, i.e., its type environment, global-symbol table, and function table belong to the sets, $valid_{\text{tenv}}$ [66, Definition 5.12], $valid_{\text{st}}$ [66, Definition 5.13], and $valid_{\text{ft}}$ [66, Definition 5.17], respectively, and

- translatable, i.e., belong to the set $xltbl_{\text{prog}}$ [66, Definition 7.41].

The formal linking operator $link_\pi$ was designed with an idea in mind to link arbitrarily many programs. That is why linking of two programs does not deliver us a program which respects the two mentioned statements simply by construction.

Validity

Let us analyze which preconditions two given programs must fulfill in order to respect the correctness requirements to a linker. At first glance it seems that the two programs subject to linking must be valid. However, that a program is supposed to be linked means it has external functions with empty bodies. These functions at least do not satisfy a property that their last statement is *return*, which is a part of the valid function-table definition. As for type environments and global-symbol tables, we can assume their validity before linking.

Besides the validity of type environments te_1 and te_2, we need that there are no two different names in both environments that describe the same type and vice versa: if in both tables the same name is used, then the types behind this name are the same:

$$dstnct_{\text{tenv}}(te_1, te_2) \stackrel{\text{def}}{=} \forall\, t_1 \in te_1 :\ \forall\, t_2 \in te_2 :\ t_1.tn = t_2.tn \longleftrightarrow t_1.td = t_2.td.$$

Altogether, the preconditions to linking of type environments are:

$$\textit{precond-link}_{\text{te}}(te_1, te_2) \stackrel{\text{def}}{=} te_1 \in valid_{\text{tenv}} \ \wedge\ te_2 \in valid_{\text{tenv}} \ \wedge\ dstnct_{\text{tenv}}(te_1, te_2).$$

The following lemma claims correctness of type environments linking.

Lemma 6.9 (Linking preserves validity of type environment) Assume that the preconditions to linking of type environments hold for te_1 and te_2, then the linked type environment is valid:

$$\textit{precond-link}_{\text{te}}(te_1, te_2) \longrightarrow link_{\text{te}}(te_1, te_2) \in valid_{\text{tenv}}$$

Isabelle: `cvm/linker/linker_tt_props.valid_tenv_link_tt`

In order to obtain a valid symbol table after linking it is sufficient to have a weaker precondition compared to type environments: if two variables with the same name are used in both original symbol table, then these variables have the same type:

$$dstnct_{\text{st}}(st_1, st_2) \stackrel{\text{def}}{=} \forall\, s_1 \in st_1 :\ \forall\, s_2 \in st_2 :\ s_1.vn = s_2.vn \longrightarrow s_1.ty = s_2.ty.$$

Combining this with the validity notion we obtain the preconditions to linking of symbol tables:

$$\begin{aligned}\textit{precond-link}_{\text{st}}(te_1, st_1, te_2, st_2) &\stackrel{\text{def}}{=} \\ & st_1 \in valid_{\text{st}}(te_1) \ \wedge\ st_2 \in valid_{\text{st}}(te_2) \ \wedge\ dstnct_{\text{st}}(st_1, st_2).\end{aligned}$$

The next lemma is our correctness statement for linking of symbol tables.

Lemma 6.10 (Linking preserves symbol-table validity) Assume that the preconditions to linking of symbol tables hold for st_1 and st_2 with respect to type environments te_1 and te_2, then the linked symbol table is valid:

$$\textit{precond-link}_{\text{st}}(te_1, st_1, te_2, st_2) \longrightarrow link_{\text{st}}(st_1, st_2) \in valid_{\text{st}}(link_{\text{te}}(te_1, te_2))$$

Isabelle: `cvm/linker/linker_st_props.valid_symboltalbe_link_st`

For a correct linking of function tables ft_1 and ft_3 we need to assume a more strict notion of distinction. First of all, function names, i.e., the first components of each table's item, must be distinct. Moreover, all statements in both tables must be different, i.e., the predicate $dstnct_{\text{s}}^{ft}$ [66, Definition 5.16] holds for both function tables. Finally, as each undefined function is merged during linking with a corresponding defined function we need to assume that in the given function tables the names of all defined functions are distinct:

$$\begin{aligned} \textit{distinct-def-func-names}(ft_1, ft_2) \stackrel{\text{def}}{=} \quad & \{x : \exists\, i : \ \textit{rem-ext-func}(ft_1)[i].fn = x\} \\ & \cap \{x : \exists\, i : \ \textit{rem-ext-func}(ft_2)[i].fn = x\} \ = \ \emptyset. \end{aligned}$$

Putting these conditions together we obtain the definition of distinct function tables:

$$\begin{aligned} \textit{dstnct}_{\text{ft}}(ft_1, ft_2) \stackrel{\text{def}}{=} \quad & \forall\, i \neq j : \ ft_1[i].fn \neq ft_1[j].fn \\ \wedge \quad & \forall\, i \neq j : \ ft_2[i].fn \neq ft_2[j].fn \\ \wedge \quad & \textit{dstnct}_{\text{s}}^{ft}(ft_1) \ \wedge \ \textit{dstnct}_{\text{s}}^{ft}(ft_2) \\ \wedge \quad & \textit{distinct-def-func-names}(ft_1, ft_2) \end{aligned}$$

In order to ensure that in the linked function table no external, i.e., undefined, functions occur we need to have that all undefined functions from one function table are defined in the second:

$$\begin{aligned} & \textit{all-ext-func-covered}(ft_1, ft_2) \stackrel{\text{def}}{=} \\ & \qquad \forall\, f_2 \in ft_2 : \textit{is-ext-func}(f_2.fd) \\ & \qquad\qquad \longrightarrow \ \exists\, f_1 \in ft_1 : \ f_1.fn = f_2.fn \ \wedge \ \neg \textit{is-ext-func}(f_1.fd). \end{aligned}$$

The predicate $\textit{linked-calls-corr}_{\text{ft}}(ft)$ ensures that each call statement in (each function of) the function table ft is appropriately defined: calls are made to the functions with defined bodies whereas external calls are used to invoke undefined functions. The predicate checks every single statement s of ft with the auxiliary predicate $\textit{linked-calls-corr}_{\text{stmt}}$:

$$\textit{linked-calls-corr}_{\text{ft}}(ft) \stackrel{\text{def}}{=} \ ft',$$

$$ft'[i].fd.body \ = \ \textit{linked-calls-corr}_{\text{stmt}}(ft, ft[i].fd.body).$$

The predicate $\textit{linked-calls-corr}_{\text{stmt}}(ft, s)$ is defined by induction over the statement structure. For those statements which have substatements s', namely comp, ifte, and loop, we apply $\textit{linked-calls-corr}_{\text{ft}}(ft, s')$ recursively, i.e.,

$$\textit{linked-calls-corr}_{\text{stmt}}(ft, \mathsf{loop}(e, s', sid)) \stackrel{\text{def}}{=} \textit{linked-calls-corr}_{\text{stmt}}(ft, s').$$

For the call or external call statements to the function of name fn we find its body fd in the function table ft. We check $\neg \textit{is-ext-func}(fd)$ for call statements and $\textit{is-ext-func}(fd)$ for external calls, respectively.

$$\begin{aligned} & \textit{linked-calls-corr}_{\text{stmt}}(ft, \mathsf{sCall}(e, fn, es, sid)) \stackrel{\text{def}}{=} \exists\, fd : \ (fn, fd) \in ft \wedge \neg \textit{is-ext-func}(fd), \\ & \textit{linked-calls-corr}_{\text{stmt}}(ft, \mathsf{esCall}(e, fn, es, sid)) \stackrel{\text{def}}{=} \exists\, fd : \ (fn, fd) \in ft \wedge \textit{is-ext-func}(fd). \end{aligned}$$

For all remaining statements the predicate returns true.

We combine *all-ext-func-covered* and $\mathit{linked\text{-}calls\text{-}corr}_{\mathrm{ft}}$ with the C0 functions validity requirement $\mathit{valid}_{\mathrm{fun}}$ [66, Definition 5.14] for all non-external functions in both function tables in order to obtain the preconditions to correct linking of function tables:

$$\begin{aligned}
&\mathit{precond\text{-}link}_{\mathrm{ft}}(te_1, st_1, ft_1, te_2, st_2, ft_2) \stackrel{\mathrm{def}}{=} \\
&\qquad \mathit{all\text{-}ext\text{-}func\text{-}covered}(ft_1, ft_2) \ \wedge \ \mathit{all\text{-}ext\text{-}func\text{-}covered}(ft_2, ft_1) \\
&\wedge \quad \mathit{linked\text{-}calls\text{-}corr}_{\mathrm{ft}}(ft_1) \ \wedge \ \mathit{linked\text{-}calls\text{-}corr}_{\mathrm{ft}}(ft_2) \\
&\wedge \quad \forall\, f \in \mathit{rem\text{-}ext\text{-}func}(ft_1) : \ f.fd \in \mathit{valid}_{\mathrm{fun}}(te_1, ft_1, st_1) \\
&\wedge \quad \forall\, f \in \mathit{rem\text{-}ext\text{-}func}(ft_2) : \ f.fd \in \mathit{valid}_{\mathrm{fun}}(te_2, ft_2, st_2).
\end{aligned}$$

The following lemma justifies the correctness of function-tables linking. Besides the described preconditions and distinction requirements it has an additional non-trivial assumption. In order to guarantee that all call statements are valid in the linked table we have to assume that the functions which occur in both original function tables have the same signature:

$$\begin{aligned}
&\mathit{same\text{-}signatures}(ft_1, ft_2) \stackrel{\mathrm{def}}{=} \\
&\quad \forall\, f_1 \in ft_1 : \ \forall\, f_2 \in ft_2 : \ f_1.fn = f_2.fn \ \longrightarrow \ f_1.fd.params = f_2.fd.params \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad f_2.fd.rtype = f_2.fd.rtype.
\end{aligned}$$

Lemma 6.11 (Linking establishes function-table validity) Let te_1 and te_2 be type environments, let ft_1 and ft_2 be function tables, and let st_1 and st_2 be symbol tables. Assume that (1) the preconditions to linking of function tables hold, (2) the first function table is not empty and the functions that occur in both tables have same signatures, and (3) the type environments, function tables, and symbol tables are pairwise distinct, then the linked function table is valid:

$$\begin{aligned}
&(1) \qquad \mathit{precond\text{-}link}_{\mathrm{ft}}(te_1, st_1, ft_1, te_2, st_2, ft_2) \\
&(2) \quad \wedge \quad ft_1 \neq [] \ \wedge \ \mathit{same\text{-}signatures}(ft_1, ft_2) \\
&(3) \quad \wedge \quad \mathit{dstnct}_{\mathrm{tenv}}(te_1, te_2) \ \wedge \ \mathit{dstnct}_{\mathrm{st}}(st_1, st_2) \ \wedge \ \mathit{dstnct}_{\mathrm{ft}}(ft_1, ft_2) \\
&\qquad \longrightarrow \quad \mathit{link}_{\mathrm{ft}}(ft_1, ft_2) \in \mathit{valid}_{\mathrm{ft}}(\mathit{link}_{\mathrm{te}}(te_1, te_2), \mathit{link}_{\mathrm{st}}(st_1, st_2))
\end{aligned}$$

Isabelle: `cvm/linker/linker_pt_props.link_pt_in_valid_proctables`

Proof. Let us abbreviate $te = \mathit{link}_{\mathrm{te}}(te_1, te_2)$, $ft = \mathit{link}_{\mathrm{ft}}(ft_1, ft_2)$, and $gst = \mathit{link}_{\mathrm{st}}(st_1, st_2)$. We unfold the function-table validity notion [66, Definition 5.17] and need to prove the four following facts about the linked function table.

Subgoal 1. $\forall\, f \in ft : \ f.fd \in \mathit{valid}_{\mathrm{fun}}(te, ft, gst)$: all functions constituting the function table are valid. Assumption (1) comprises the statements $\mathit{all\text{-}ext\text{-}func\text{-}covered}(ft_1, ft_2)$ and $\mathit{all\text{-}ext\text{-}func\text{-}covered}(ft_2, ft_1)$ which ensure that there is no external, i.e., undefined, functions in the linked function table. The linking operator affects only call statements. From $\mathit{same\text{-}signatures}(ft_1, ft_2)$ we conclude that all call statements stay valid.

Subgoal 2. $\forall\, i \neq j : \ ft[i].fn \neq ft[j].fn$: all function names are distinct. From the last term of (3) we have that all names in the original function tables are distinct as well

as that all defined functions in both tables have different names. It is enough to conclude the subgoal because the linking algorithm merges all undefined functions into one entry.

Subgoal 3. $dstnct_{\mathrm{s}}^{ft}(ft)$: all statements are distinct. From $dstnct_{\mathrm{ft}}(ft_1, ft_2)$ we have that statements of original programs are distinct. The linking algorithm involves the statement renumbering mechanism. This ensures that all statements besides calls are distinct in the obtained function table. As for the call statements, the lemma's assumptions $\textit{linked-calls-corr}_{\mathrm{ft}}(ft_1)$ and $\textit{linked-calls-corr}_{\mathrm{ft}}(ft_2)$ forbid situations where in one of the original function tables, say ft_1, there are call and external call statements with the same statement identifier and signature which are transformed during linking into identical call statements by means of the function $\textit{ext-upd}_{\mathrm{stmt}}$ (Definition 6.3).

Subgoal 4. $ft \neq []$: the function table is not empty. Follows from $ft_1 \neq []$.

□

Translatability

In order to formulate a correctness statement that linked programs are translatable let us recall some definitions from the C0 small-step semantics and code generation algorithm. We denote that a function f is translatable by $f \in \textit{xltbl}_{\mathrm{func}}(te, ft, gst)$ [66, Definition 7.40]. The allocated size of a symbol table st is computed by means of the function $\textit{asize}_{\mathrm{st}}(st)$ [66, Definition 7.11]. The code size of a function table ft with respect to a type environment te and global-symbol table gst is computed by means of the function $\textit{csize}_{\mathrm{prog}}(te, gst, ft)$ [66, Definition 7.5].

Lemma 6.12 (Linking produces translatable programs) Let te_1 and te_2 be type environments, let ft_1 and ft_2 be function tables, and let st_1 and st_2 be symbol tables. Let $ft_1' = \textit{ext-upd}_{\mathrm{ft}}(ft_2, ft_1)$ and $ft_2' = \textit{ext-upd}_{\mathrm{ft}}(ft_1, ft_2)$ be updated function tables where all external calls are replaced. Assume that (1) preconditions to linking of type environments, symbol tables, and function tables hold, (2) all functions in updated function tables are translatable, (3) doubled size of symbol tables sum fits into 32 bits, and (4) computed size of updated function tables fits into 26 bits with respect to the program base, then the linked program is translatable:

$$
\begin{array}{lll}
(1) & & \textit{precond-link}_{\mathrm{te}}(te_1, te_2) \\
 & \wedge & \textit{precond-link}_{\mathrm{st}}(te_1, st_1, te_2, st_2) \\
 & \wedge & \textit{precond-link}_{\mathrm{ft}}(te_1, st_1, ft_1, te_2, st_2, ft_2) \\
(2) & \wedge & \forall\, f \in ft_1' :\ f.fd \in \textit{xltbl}_{\mathrm{func}}(te_1, ft_1, st_1) \\
 & \wedge & \forall\, f \in ft_2' :\ f.fd \in \textit{xltbl}_{\mathrm{func}}(te_2, ft_2, st_2) \\
(3) & \wedge & 2 \cdot (\textit{asize}_{\mathrm{st}}(st_1) + \textit{asize}_{\mathrm{st}}(st_2)) < 2^{32} \\
(4) & \wedge & \texttt{PROGBASE} + 4 \cdot (\textit{csize}_{\mathrm{prog}}(te_1, st_1, ft_1') + \textit{csize}_{\mathrm{prog}}(te_2, st_2, ft_2')) < 2^{25} \\
 & \longrightarrow & (\textit{link}_{\mathrm{te}}(te_1, te_2), \textit{link}_{\mathrm{ft}}(ft_1, ft_2), \textit{link}_{\mathrm{st}}(st_1, st_2)) \in \textit{xltbl}_{\mathrm{prog}}.
\end{array}
$$

Isabelle: `cvm/linker/linker_transl_props.link_in_translatable_programs`

Proof. Let us abbreviate $te = link_{\text{te}}(te_1, te_2)$, $ft = link_{\text{ft}}(ft_1, ft_2)$, and $gst = link_{\text{st}}(st_1, st_2)$. We unfold the definition of translatable programs [66, Definition 7.41] and need to prove the three following facts.

Subgoal 1. $\texttt{PROGBASE} + 4 \cdot csize_{\text{prog}}(te, gst, ft) < 2^{25}$: program base plus the code size fits into 26 bits. Follows from assumption (4).

Subgoal 2. $\forall\ f \in ft :\ f.fd \in xltbl_{\text{func}}(te, ft, gst)$: all functions are translatable. In (2) we assume for both function tables that all functions are translatable after replacing all external calls. Since the linking algorithm replaces all external calls we conclude the subgoal.

Subgoal 3. $2 \cdot asize_{\text{st}}(gst) < 2^{32}$: global-symbol table is translatable. Follows from assumption (3) and the fact that the size of the linked symbol table is always less than the sum of sizes of given symbol tables.

Subgoal 4. $abase_{\text{lm}}(te, ft, gst) < 2^{32}$: stack start address fits into 32 bits. Follows from the definition of the constant $\texttt{ABASE}_{\text{lm}}$ which is equal to the stack start address.

Subgoal 5. $\texttt{ABASE}_{\text{hm}} + \texttt{ASIZE}_{\text{hm}}^{\max} < 2^{32}$: heap end address fits into 32 bits. Follows from definition of the constants.

□

6.3 Obtaining a Concrete Kernel

Having the linking mechanism formally defined we can formalize a concrete kernel as a function of an abstract kernel. The concrete kernel is the CVM framework implementation linked with an abstract kernel. Let π_{AK} be a C0 program of the abstract kernel.

The C0 program of the concrete kernel is defined as:

$$\pi_{\text{CK}} :: \Pi_{\text{C0}} \mapsto \Pi_{\text{C0}},$$

$$\pi_{\text{CK}}(\pi_{\text{AK}}) \stackrel{\text{def}}{=} link_{\pi}(\pi_{\text{CVM}}, \pi_{\text{AK}}).$$

Additionally we define three functions for accessing the type environment, function table, and global symbol table of the concrete kernel. The type environment of the concrete kernel is accessed with the following function:

$$te_{\text{CK}} :: \Pi_{\text{C0}} \mapsto Tenv,$$

$$te_{\text{CK}}(\pi_{\text{AK}}) \stackrel{\text{def}}{=} \pi_{\text{CK}}(\pi_{\text{AK}}).te.$$

We define the function for extracting the function table of the concrete kernel:

$$ft_{\text{CK}} :: \Pi_{\text{C0}} \mapsto Functable,$$

$$ft_{\text{CK}}(\pi_{\text{AK}}) \stackrel{\text{def}}{=} \pi_{\text{CK}}(\pi_{\text{AK}}).ft.$$

Finally, the global symbol table of the concrete kernel is obtained by means of the following function:

$$gst_{\text{CK}} :: \Pi_{\text{C0}} \mapsto Symtable,$$

$$gst_{\text{CK}}(\pi_{\text{AK}}) \stackrel{\text{def}}{=} \pi_{\text{CK}}(\pi_{\text{AK}}).gst.$$

6.4 Validity and Translatability of the Concrete Kernel

For application of the linker correctness theorem some properties have to hold for each program. These properties could be shown for the CVM implementation because we have the completely defined CVM program. As for the abstract kernel, the properties will be shown after the instantiation of π_{AK} with particular code. Thus, these properties will stay as assumptions to the top level theorem. We define predicate

$$\mathit{abs\text{-}kernel\text{-}props} :: \Pi_{\mathrm{C0}} \mapsto \mathbb{B}.$$

where we collect all necessary properties for abstract kernel.

Validity of a linked type environment. Validity of the CVM type environment is shown by construction.

Corollary 6.13 $\mathit{te}_{\mathrm{CVM}} \in \mathit{valid}_{\mathrm{tenv}}$.

Isabelle: `cvm/code/cvm_tt_props.valid_tenv_cvm_tt`

Other preconditions to correct linking of type environments could not be shown at the moment. We add them to the properties of the abstract kernel:

$$\begin{array}{lcl} \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) & \stackrel{\mathrm{def}}{=} & \pi_{\mathrm{AK}}.\mathit{te} \in \mathit{valid}_{\mathrm{tenv}} \\ & \wedge & \mathit{dstnct}_{\mathrm{tenv}}(\mathit{te}_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.\mathit{te}) \\ & \wedge & \cdots \end{array}$$

Having this, and using Lemma 6.9 we can show the validity of the linked type environment.

Lemma 6.14 (Validity of the concrete kernel type environment) Assume that the properties of the abstract kernel hold, then the type environment of the concrete kernel is valid:

$$\mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \longrightarrow \mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}) \in \mathit{valid}_{\mathrm{tenv}}.$$

Isabelle: `cvm/config/c0_config_props.abs_kernel_properties_impl_valid_tenv_linked_tt`

Validity of a linked symbol table. Similarly to the type environment, the validity of the CVM symbol table is shown by construction.

Corollary 6.15 $\mathit{gst}_{\mathrm{CVM}} \in \mathit{valid}_{\mathrm{st}}(\mathit{te}_{\mathrm{CVM}})$.

Isabelle: `cvm/code/cvm_st_props.valid_symboltable_cvm`

From the source code of VAMOS and OLOS we know that the CVM code and the abstract-kernel code do not have shared global variables. This allows us to strengthen $\mathit{dstnct}_{\mathrm{st}}$:

$$\begin{array}{lcl} \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) & \stackrel{\mathrm{def}}{=} & \cdots \\ & \wedge & \pi_{\mathrm{AK}}.\mathit{gst} \in \mathit{valid}_{\mathrm{st}}(\pi_{\mathrm{AK}}.\mathit{te}) \\ & \wedge & \mathit{vns}(\mathit{gst}_{\mathrm{CVM}}) \cap \mathit{vns}(\pi_{\mathrm{AK}}.\mathit{gst}) = \emptyset \\ & \wedge & \cdots \end{array}$$

Since we can show the implication:

$$vns(gst_{\mathrm{CVM}}) \cap vns(\pi_{\mathrm{AK}}.gst) = \emptyset \longrightarrow dstnct_{\mathrm{st}}(gst_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.gst),$$

the validity of the linked symbol table follows from Lemma 6.10.

Lemma 6.16 (Validity of the concrete kernel symbol table) Assume that the properties of the abstract kernel hold, then the symbol table of the concrete kernel is valid:

$$\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \longrightarrow gst_{\mathrm{CK}}(\pi_{\mathrm{AK}}) \in valid_{\mathrm{st}}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}})).$$

Isabelle: `cvm/config/c0_config_props.abs_kernel_properties_impl_valid_symboltable_linked_st`

Validity of a linked function table. For the CVM function table we have the following properties true by construction:

Corollary 6.17 $\forall\, f \in \textit{rem-ext-func}(ft_{\mathrm{CVM}}) :$
$f.fn \in valid_{\mathrm{fun}}(te_{\mathrm{CVM}}, ft_{\mathrm{CVM}}, gst_{\mathrm{CVM}})$.

Isabelle: `cvm/code/cvm_pt_props.list_all_valid_functions_filter_not_is_Skip_cvm_pt`

Corollary 6.18 $ft_{\mathrm{CVM}} \neq []$.

Isabelle: `cvm/code/cvm_pt_props.cvm_pt_not_Nil`

Corollary 6.19 $\forall\, i \neq j :\ ft_{\mathrm{CVM}}[i].fn \neq ft_{\mathrm{CVM}}[j].fn.$

Isabelle: `cvm/code/cvm_pt_props.distinct_map_fst_cvm_pt`

Corollary 6.20 $dstnct_{\mathrm{s}}^{ft}(ft_{\mathrm{CVM}})$.

Isabelle: `cvm/code/cvm_pt_props.stmts_distinct_pt_cvm_pt`

Corollary 6.21 $\textit{linked-calls-corr}_{\mathrm{ft}}(ft_{\mathrm{CVM}})$.

Isabelle: `cvm/code/cvm_pt_props.correct_kind_of_Call_cvm_pt`

For the abstract kernel function table (possibly together with the CVM function tables) we reserve the following statements to be proven later:

$$\begin{aligned}
\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \stackrel{\mathrm{def}}{=}\ & \cdots \\
\wedge\ & \textit{same-signatures}(ft_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.ft) \\
\wedge\ & \forall\, i \neq j :\ \pi_{\mathrm{AK}}.ft[i].fn \neq \pi_{\mathrm{AK}}.ft[j].fn \\
\wedge\ & dstnct_{\mathrm{s}}^{ft}(\pi_{\mathrm{AK}}.ft) \\
\wedge\ & \textit{all-ext-func-covered}(ft_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.ft) \\
\wedge\ & \textit{all-ext-func-covered}(\pi_{\mathrm{AK}}.ft, ft_{\mathrm{CVM}}) \\
\wedge\ & \textit{linked-calls-corr}_{\mathrm{ft}}(\pi_{\mathrm{AK}}.ft) \\
\wedge\ & \textit{distinct-def-func-names}(ft_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.ft) \\
\wedge\ & \forall\, f \in \textit{rem-ext-func}(\pi_{\mathrm{AK}}.ft) : \\
& \qquad f.fd \in valid_{\mathrm{fun}}(\pi_{\mathrm{AK}}.te, \pi_{\mathrm{AK}}.ft, \pi_{\mathrm{AK}}.gst) \\
\wedge\ & \cdots
\end{aligned}$$

To prove the validity of the linked function table we show first:

$$\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \longrightarrow \textit{precond-link}_{\mathrm{ft}}(te_{\mathrm{CVM}}, gst_{\mathrm{CVM}}, ft_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.te, \pi_{\mathrm{AK}}.gst, \pi_{\mathrm{AK}}.ft)$$

and

$$\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \longrightarrow \textit{dstnct}_{\mathrm{ft}}(ft_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.ft),$$

and with the help of Lemma 6.11 conclude with the following lemma.

Lemma 6.22 (Validity of the concrete kernel function table) Assume that the properties of the abstract kernel hold, then the function table of the concrete kernel is valid:

$$\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \longrightarrow ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}) \in \textit{valid}_{\mathrm{ft}}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), gst_{\mathrm{CK}}(\pi_{\mathrm{AK}})).$$

Isabelle: `cvm/config/c0_config_props.abs_kernel_properties_impl_valid_proctables`

Translatability of a linked program. It is easy to show that the CVM functions with external calls replaced are translatable.

Corollary 6.23
$$\forall\, f \in \textit{ext-upd}_{\mathrm{ft}}(\pi_{\mathrm{AK}}.ft, ft_{\mathrm{CVM}}) : f.fd \in \textit{xltbl}_{\mathrm{func}}(te_{\mathrm{CVM}}, ft_{\mathrm{CVM}}, gst_{\mathrm{CVM}})$$

Isabelle: `cvm/code/cvm_pt_props.list_all_translatable_functions_ESCalls_update_pt_2_cvm_pt`

Besides the translatability of abstract kernel functions we need some restrictions on the size of the compiled code and global symbol table:

$$\begin{aligned}
\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \stackrel{\mathrm{def}}{=}\ & \\
& \ldots \\
\wedge\ & \forall\, f \in \textit{ext-upd}_{\mathrm{ft}}(ft_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.ft) : \\
& \quad f.fd \in \textit{xltbl}_{\mathrm{func}}(\pi_{\mathrm{AK}}.te, \pi_{\mathrm{AK}}.ft, \pi_{\mathrm{AK}}.gst) \\
\wedge\ & 4 \cdot \textit{csize}_{\mathrm{prog}}(\pi_{\mathrm{AK}}.te, \pi_{\mathrm{AK}}.gst, \textit{ext-upd}_{\mathrm{ft}}(ft_{\mathrm{CVM}}, \pi_{\mathrm{AK}}.ft)) \\
& \quad + \mathtt{BUBBLE}_{\mathrm{code}} = \mathtt{ABASE}_{\mathrm{gm}} - \mathtt{PROGBASE} \\
& \quad - 4 \cdot \textit{csize}_{\mathrm{prog}}(te_{\mathrm{CVM}}, gst_{\mathrm{CVM}}, \textit{ext-upd}_{\mathrm{ft}}(\pi_{\mathrm{AK}}.ft, ft_{\mathrm{CVM}})) \\
\wedge\ & \textit{asize}_{\mathrm{st}}(\pi_{\mathrm{AK}}.gst) + \mathtt{BUBBLE}_{\mathrm{gm}} = \\
& \quad \mathtt{ABASE}_{\mathrm{lm}} - \mathtt{ABASE}_{\mathrm{gm}} - \textit{asize}_{\mathrm{st}}(gst_{\mathrm{CVM}}) \\
\wedge\ & \ldots
\end{aligned}$$

Using Lemma 6.12 we can prove that the concrete kernel program is translatable.

Lemma 6.24 (Translatability of the concrete kernel program) Assume that the properties of the abstract kernel hold, then the program of the concrete kernel is translatable:

$$\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \longrightarrow (te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}), gst_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \in \textit{xltbl}_{\mathrm{prog}}.$$

Isabelle: `cvm/config/c0_config_props.abs_kernel_properties_impl_linked_in_translatable_programs`

Chapter 7

Correctness of a Microkernel

Contents

In this chapter we introduce correctness criteria and a theorem of the CVM model, a framework for microkernel developers. The criteria are formulated as an abstraction relation from a VAMP ISA with devices configurations towards the CVM model state. The paper-and-pencil theory behind the CVM model correctness was originally introduced by Gargano et al. [41]. Besides this thesis the mentioned theory inspired also In der Rieden's thesis [32] which we discuss in Section 11.

We start in Section 7.1 with a formal introduction of correctness criteria of the CVM model. In Section 7.2 we state the correctness theorem of CVM. Finally, in Section 7.3 we outline a skeleton of this theorem's proof. Complete details on each induction case of the proof are presented in further chapters (cf. Chapter 9 and Chapter 10).

7.1 CVM Correctness Criteria

This section introduces an abstraction relation for CVM

$$\mathit{cvm\text{-}sim} :: \Pi_{\mathrm{C0}} \times C_{\mathrm{CVM}} \times C_{\mathrm{C0}} \times C_{\mathrm{ISA+DS}} \mapsto \mathbb{B},$$

$$\mathit{cvm\text{-}sim}(\pi_{\mathrm{AK}}, c_{\mathrm{CVM}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}),$$

which holds if the CVM configuration c_{CVM} is encoded by both the states of VAMP ISA with devices $c_{\mathrm{ISA+DS}}$ and the concrete kernel c_{C0} with respect to the abstract kernel program π_{AK}. The structure of the relation is depicted in Figure 7.1.

We distinguish three kinds of CVM states: (i) user executions, (ii) kernel execution during waiting for interrupts, and (iii) other kernel executions. It its simple to distinguish the first state: a CVM configuration c_{CVM} encodes some user execution if the current process component $c_{\mathrm{CVM}}.\mathit{cup}$ has a value of some *pid*. As for the kernel executions we introduce the predicate $\mathit{is\text{-}wait\text{-}state}(c_{\mathrm{CVM}})$ to denote a waiting for interrupt state. Recall from Section 5.3 that when CVM waits for interrupts the program rest of its abstract

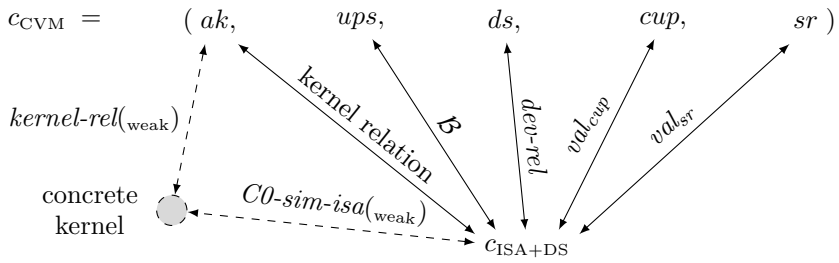

Figure 7.1: CVM abstraction relation.

kernel component $c_{\mathrm{CVM}}.ak$ is artificially set to an assembly statement. Therefore, we define

$$\textit{is-wait-state}(c_{\mathrm{CVM}}) \stackrel{\mathrm{def}}{=} \textit{is-}\mathsf{asm}(c_{\mathrm{CVM}}.ak.prog).$$

Depending on the kind of a CVM state the CVM abstraction relation is introduced in two forms: $\textit{cvm-sim}_{\mathrm{kernel}}$ and $\textit{cvm-sim}_{\mathrm{weak}}$. The former must hold for kernel configurations except for the waiting for interrupt state while the latter is designed to specify correctness criteria during user executions and when the kernel is waiting for interrupts.

$$\textit{cvm-sim}(\pi_{\mathrm{AK}}, c_{\mathrm{CVM}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}) \stackrel{\mathrm{def}}{=}$$
$$\begin{cases} \textit{cvm-sim}_{\mathrm{weak}}(\pi_{\mathrm{AK}}, c_{\mathrm{CVM}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}, 0) & \text{if} \quad c_{\mathrm{CVM}}.cup = \bot \\ & \quad \wedge \quad \textit{is-wait-state}(c_{\mathrm{CVM}}) \\ \textit{cvm-sim}_{\mathrm{kernel}}(\pi_{\mathrm{AK}}, c_{\mathrm{CVM}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}) & \text{if} \quad c_{\mathrm{CVM}}.cup = \bot \\ & \quad \wedge \quad \neg\textit{is-wait-state}(c_{\mathrm{CVM}}) \\ \textit{cvm-sim}_{\mathrm{weak}}(\pi_{\mathrm{AK}}, c_{\mathrm{CVM}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}, pid) & \text{if} \quad c_{\mathrm{CVM}}.cup = \lfloor pid \rfloor \end{cases}$$

Further in this section we formally define relations $\textit{cvm-sim}_{\mathrm{kernel}}$ and $\textit{cvm-sim}_{\mathrm{weak}}$. Before that, let us introduces a number of auxiliary definitions.

7.1.1 Obtaining Values of Variables

In order to relate configurations of the CVM model to states of VAMP ISA with devices we must be able to express such high-level concepts as the kernel's C0 variables in terms of low-level hardware models. When a kernel and user processes run on a physical combined system there are only two places to store their code and data: the main physical memory and the hard disk. Further we define functions that read values of CVM's and user's variables from these two storages.

We compute an offset in the memory of an assembly or ISA configurations for a variable of name vn within a symbol table st by means of the following function [66, Def. 7.10]:

$$\textit{displ}_{\mathrm{v}}(st, vn) \stackrel{\mathrm{def}}{=} \begin{cases} \mathsf{A} & \text{if } st = [] \\ 0 & \text{if } st = (vn, ty) \circ st' \\ \textit{asize}_{\mathrm{t}}(ty) + \textit{displ}_{\mathrm{v}}(st', vn) & \text{if } st = (vn', ty) \circ st' \ \wedge \ vn' \neq vn \end{cases}$$

Having this and assuming that vn is a name of a global variable we define the variable's address as

$$ad_{vn} \stackrel{\mathrm{def}}{=} \mathtt{ABASE}_{\mathrm{gm}} + \textit{displ}_{\mathrm{v}}(gst_{\mathrm{CVM}}, vn).$$

We extend this definition for process's registers. We know that registers of a process p are stored consecutively in its process control block. Let *reg* be a serial number of a register in the PCB we are interested in. Then the address of this register in the physical memory is computed by the following formula:

$$ad^p_{reg} \stackrel{\text{def}}{=} \texttt{ABASE}_{\text{gm}} + \mathit{displ}_{\text{v}}(\mathit{gst}_{\text{CVM}}, \mathit{pcb}) + p \cdot 2^9 + \mathit{reg} \cdot 4.$$

Above, 2^9 is a size of a single process control block. In fact, the data in a pcb consume 328 bytes. We align this size to a closest power of 2 from above — 2^9 — by adding a respective amount of empty space at the end of each pcb.

We read a bit vector from an ISA memory m at an address a given as a natural number by means of the function

$$\mathit{read\text{-}isa}(m, a) \stackrel{\text{def}}{=} m_{\mathit{word}}(\mathit{bin}(a)).$$

Now we have a possibility to interpret the resulting bit vector of such reading either as a binary or a two's complement number and convert it into a natural or an integer number, respectively. For this we define two functions which read the memory of an ISA configuration c_{ISA} and perform the desired conversions:

$$\begin{aligned} \mathit{nat}_{\text{vars}}(c_{\text{ISA}}, a) &\stackrel{\text{def}}{=} \langle \mathit{read\text{-}isa}(c_{\text{ISA}}.\mathit{Mem}_{\text{ISA}}, a) \rangle, \\ \mathit{int}_{\text{vars}}(c_{\text{ISA}}, a) &\stackrel{\text{def}}{=} [\mathit{read\text{-}isa}(c_{\text{ISA}}.\mathit{Mem}_{\text{ISA}}, a)]. \end{aligned}$$

Combining these functions with our notation for a variable addresses we define aliases for obtaining values of particular CVM implementation variables. The value of the current process variable `cup` is extracted from the memory of a VAMP ISA with devices configuration $c_{\text{ISA+DS}}$ by means of the function

$$\mathit{val}_{\mathit{cup}}(c_{\text{ISA+DS}}) \stackrel{\text{def}}{=} \mathit{nat}_{\text{vars}}(c_{\text{ISA+DS}}.\mathit{cpu}, \mathit{ad}_{\mathit{cup}}),$$

while the value of the status register variable `sr` is obtained as

$$\mathit{val}_{\mathit{sr}}(c_{\text{ISA+DS}}) \stackrel{\text{def}}{=} \mathit{nat}_{\text{vars}}(c_{\text{ISA+DS}}.\mathit{cpu}, \mathit{ad}_{\mathit{sr}}).$$

With the help of these two definitions we easily will be able to formulate correctness relations for the corresponding components of the CVM model: we will equate the values retrieved from ISA to the respective values $c_{\text{CVM}}.\mathit{cup}$ and $c_{\text{CVM}}.\mathit{sr}$.

Recall that the swap hard disk is a device with identifier `SWAP_DID` in the devices system configuration $c_{\text{ISA+DS}}.\mathit{devs}$. We read an integer value from the hard disk's swap memory at an address a given as a natural 32-bit number by means of the function:

$$\mathit{int}_{\text{swap}}(c_{\text{DS}}, a) \stackrel{\text{def}}{=} \mathit{n2i}(\mathit{the\text{-}hd}(c_{\text{DS}}(\texttt{SWAP\_DID})).\mathit{sm}[150 \cdot 1024 + a/4]).$$

Note that an offset of 150 pages appears in the swap memory address computation due to the boot region located at the beginning of the hard disk.

7.1.2 Devices Relation

The first non-trivial, but nevertheless simple, correctness criterion of CVM is the devices relation. Recall Definition 5.6: on the initialization CVM's devices system is constructed as a copy of VAMP ISA's devices system from which the swap hard disk is excluded.

By design the remaining devices operate equally in CVM configurations and in the underlying physical combined system configuration.

Definition 7.1 (Devices relation) The devices relation claims that the device states in CVM and VAMP ISA configurations are equal except for the swap hard disk which is an idle device in CVM:

$$\begin{aligned} &\textit{dev-rel}(c_{\mathrm{CVM}}, c_{\mathrm{ISA+DS}}) \stackrel{\mathrm{def}}{=} \\ &\qquad \forall\ did:\ did \neq \texttt{SWAP\_DID} \longrightarrow c_{\mathrm{CVM}}.ds(did) = c_{\mathrm{ISA+DS}}.devs(did) \\ &\quad \wedge \quad \textit{is-idle-dev}(c_{\mathrm{CVM}}.ds(\texttt{SWAP\_DID})). \end{aligned}$$

Isabelle: `cvm/cvm_correct/cvm_sim.dev_sim`

7.1.3 Relation for User Processes

The correctness relation for user processes states whether the user processes configurations $c_{\mathrm{CVM}}.ups$ are represented in the configuration $c_{\mathrm{ISA+DS}}$ of a VAMP ISA machine with devices. An active user process, i.e., the one that is currently running on the processor, is represented by the contents of hardware registers. Suspended user processes are implemented by the process control blocks. In order to distinguish these two cases we will analyze the value of the current process identifier together with the system mode predicate:

$$\textit{is-sys}(c_{\mathrm{ISA+DS}}) \stackrel{\mathrm{def}}{=} \textit{is-sys-mode}_{\mathrm{ISA}}(c_{\mathrm{ISA+DS}}.cpu).$$

In both cases memories of user processes are stored in the physical memory and on the hard disk.

We introduce the function $vm(c_{\mathrm{ISA+DS}}, p)$ which performs the mentioned case distinction and reconstructs a virtual assembly machine for a given process identifier p:

$$vm(c_{\mathrm{ISA+DS}}, p) \stackrel{\mathrm{def}}{=} \begin{cases} vm_{\mathrm{direct}}(c_{\mathrm{ISA+DS}}, p) & \text{if } \neg \textit{is-sys}(c_{\mathrm{ISA+DS}}) \ \wedge\ \textit{val}_{\textit{cup}}(c_{\mathrm{ISA+DS}}) = p \\ vm_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p) & \text{otherwise} \end{cases}.$$

The first case corresponds to a situation when the process p is active and its virtual machine is constructed directly from the hardware registers with the help of the function vm_{direct}. In the second case the process p is suspended an its virtual machine is reconstructed from variables (PCBs) by means of the function vm_{vars}. The definitions of both functions follow. Note that for reconstruction of the memory they will use the function $mem(c_{\mathrm{ISA+DS}}, p)$ which we define later.

Virtual Machine Reconstructed from Variables

Normal and delayed program counters of a process p are stored in the physical memory at addresses ad^p_{PC} and ad^p_{DPC}, respectively. In order to retrieve their values we define two following functions:

$$\begin{aligned} dpc_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p) &\stackrel{\mathrm{def}}{=} nat_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}.cpu, ad^p_{DPC}), \\ pc'_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p) &\stackrel{\mathrm{def}}{=} nat_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}.cpu, ad^p_{PC}). \end{aligned}$$

A (part of a) general purpose register file is obtained by reading 31 words from the main memory starting at address $ad^p_{GPRS_1}$. The function for reading yields a list of integers and has the following definition:

$$\mathit{get\text{-}gpr}(c_{\mathrm{ISA}}, p) \stackrel{\mathrm{def}}{=} [\mathit{int}_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad^p_{GPRS_1}), \ldots, \mathit{int}_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad^p_{GPRS_{\mathrm{EF}+n-1}})].$$

All user process have their very first general purpose register always equal to zero. Combining this we obtain all GPR values constructed from variables:

$$\mathit{gpr}_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p) \stackrel{\mathrm{def}}{=} 0 \circ \mathit{get\text{-}gpr}(c_{\mathrm{ISA+DS}}.\mathit{cpu}, p).$$

Analogous functions are defined for reading from the special purpose register file stored in process control blocks. The indices of the SPRs we are interested in are $SPRS_{pto}$ and $SPRS_{ptl}$, and the corresponding register addresses are $ad^p_{SPRS_{pto}}$ and $ad^p_{SPRS_{ptl}}$, respectively. The formal definition of the function for reading these two special registers from PCBs is as follows:

$$\mathit{get\text{-}spr}(c_{\mathrm{ISA}}, p) \stackrel{\mathrm{def}}{=} [\mathit{int}_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad^p_{SPRS_{pto}}), \mathit{int}_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad^p_{SPRS_{ptl}})].$$

Values of some special registers are not stored in the process control blocks. One example is the status registers: within CVM it is shared between all processes and its value is stored in the variable `sr`. Another example is the mode register. As user operate only in user mode it makes no sense to store its value in PCBs. Considering these facts we define the following function which delivers an assembly special purpose register file for a given user process:

$$\mathit{spr}_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p) \stackrel{\mathrm{def}}{=} \mathit{n2i}(\mathit{val}_{sr}(c_{\mathrm{ISA+DS}})) \circ 0^8 \circ \mathit{get\text{-}spr}(c_{\mathrm{ISA+DS}}.\mathit{cpu}, p) \circ 0^5 \circ 1 \circ 0^{15}.$$

Altogether, we define the virtual machine reconstructed from variables as follows:

$$\begin{aligned} \mathit{vm}_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p) \stackrel{\mathrm{def}}{=} (&\mathit{dpc}_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p),\\ &\mathit{pc}'_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p),\\ &\mathit{gpr}_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p),\\ &\mathit{spr}_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p),\\ &\mathit{mem}(c_{\mathrm{ISA+DS}}, p)). \end{aligned}$$

Virtual Machine Reconstructed Directly

When we reconstruct a virtual machine for a process p directly from the hardware registers values we need much less effort. Basically, the virtual machine's components are obtained by interpreting corresponding hardware registers as binary or two's complement numbers. For program counters we can perform a conversion to binary numbers directly and define:

$$\begin{aligned} \mathit{dpc}_{\mathrm{direct}}(c_{\mathrm{ISA+DS}}) &\stackrel{\mathrm{def}}{=} \langle c_{\mathrm{ISA+DS}}.\mathit{cpu.dpc}\rangle,\\ \mathit{pc}'_{\mathrm{direct}}(c_{\mathrm{ISA+DS}}) &\stackrel{\mathrm{def}}{=} \langle c_{\mathrm{ISA+DS}}.\mathit{cpu.pc}\rangle. \end{aligned}$$

For a VAMP ISA register file *regs* we define a function which converts the whole file to a list of integers:

$$\mathit{regs\text{-}convert}(\mathit{regs}) \stackrel{\mathrm{def}}{=} [[\mathit{regs}(\mathit{bin}(0))], \ldots, [\mathit{regs}(\mathit{bin}(31))]].$$

Having this, we define the directly reconstructed general and special purpose register files:

$$\begin{aligned} gpr_{\text{direct}}(c_{\text{ISA+DS}}) &\stackrel{\text{def}}{=} \textit{regs-convert}(c_{\text{ISA+DS}}.\textit{cpu.gpr}), \\ spr_{\text{direct}}(c_{\text{ISA+DS}}) &\stackrel{\text{def}}{=} \textit{regs-convert}(c_{\text{ISA+DS}}.\textit{cpu.spr}). \end{aligned}$$

Altogether, we define the virtual machine reconstructed directly as follows:

$$\begin{aligned} vm_{\text{direct}}(c_{\text{ISA+DS}}, p) \stackrel{\text{def}}{=} (&dpc_{\text{direct}}(c_{\text{ISA+DS}}, p), \\ &pc'_{\text{direct}}(c_{\text{ISA+DS}}, p), \\ &gpr_{\text{direct}}(c_{\text{ISA+DS}}, p), \\ &spr_{\text{direct}}(c_{\text{ISA+DS}}, p), \\ &mem(c_{\text{ISA+DS}}, p)). \end{aligned}$$

Reconstructing Virtual Memory

Independently of whether a process's virtual machine is constructed from variables or directly the process's virtual memory is stored in the main physical memory and on the hard disk. For each memory address, the decision where the data can be found is taken according to the valid bit of the respective page table entry.

On the input we have a process identifier p and a virtual address *va*. Our goal is to formally specify address translation mechanism following its implementation in CVM. First, we define how a physical memory address is computed, then we introduce a swap memory address computation.

Physical memory address. Since a single memory page in our system contains 2^{12} bytes a page and byte indices of a virtual address *va* represented as a natural number are defined as follows:

$$\begin{aligned} px(va) &\stackrel{\text{def}}{=} va/2^{12}, \\ bx(va) &\stackrel{\text{def}}{=} va \bmod 2^{12}. \end{aligned}$$

The page table origin and length for a process p are obtained from the memory of ISA by reading the registers *pto* or *ptl*, respectively, from the process control block:

$$\begin{aligned} \textit{get-pto}(c_{\text{ISA}}, p) &\stackrel{\text{def}}{=} nat_{\text{vars}}(c_{\text{ISA}}, ad^p_{SPRS_{pto}}), \\ \textit{get-ptl}(c_{\text{ISA}}, p) &\stackrel{\text{def}}{=} int_{\text{vars}}(c_{\text{ISA}}, ad^p_{SPRS_{ptl}}). \end{aligned}$$

Note that in case of *ptl* we follow the CVM's implementation and represent the result as an integer. The reason is that we want the domain of origins to be extended with -1 which denotes that a process has no memory. The i-th page table entry of a process p is delivered by reading the i-th word in the physical memory starting from the respective page table origin:

$$\textit{get-pte}(c_{\text{ISA}}, p, i) \stackrel{\text{def}}{=} nat_{\text{vars}}(c_{\text{ISA}}, \textit{get-pto}(c_{\text{ISA}}, p) \cdot 2^{12} + i \cdot 4).$$

The page index, i.e., twenty most significant bits, of a page table entry has a meaning of a physical page index. Combining it with a byte index of the virtual address we get the physical memory address. Formally:

$$\textit{pma}(c_{\text{ISA}}, p, va) \stackrel{\text{def}}{=} px(\textit{get-pte}(c_{\text{ISA}}, p, px(va))) \cdot 2^{12} + bx(va).$$

Swap memory address. We use big pages of size 2^{22} bytes. Therefore, big-page and big-byte indices of a virtual address va represented as a natural number are defined as follows:

$$\begin{aligned} bpx(va) &\stackrel{\text{def}}{=} va/2^{22}, \\ bbx(va) &\stackrel{\text{def}}{=} va \bmod 2^{22}. \end{aligned}$$

We obtain the big-page table origin for a process p by reading a natural number from the memory of ISA with devices at address ad^p_{bpto}:

$$\textit{get-bpto}(c_{\text{ISA}}, p) \stackrel{\text{def}}{=} nat_{\text{vars}}(c_{\text{ISA}}, ad^p_{bpto}).$$

Note, that big-page table origins do not store absolute values of addresses, but only indices within the *bptspace* array. To obtain the absolute address we need to add the start address of this array $ad_{bptspace}$. The i-th big-page table entry of a process p is defined as the i-th word read from the physical memory starting from the corresponding big-page table origin:

$$\textit{get-bpte}(c_{\text{ISA}}, p, i) \stackrel{\text{def}}{=} nat_{\text{vars}}(c_{\text{ISA}}, ad_{bptspace} + \textit{get-bpto}(c_{\text{ISA}}, p) \cdot 4 + i \cdot 4).$$

Big-page table entries store big-page indices. Therefore, the desired swap memory address is defined as a combination of the respective big-page table entry and the big-byte index:

$$sma(c_{\text{ISA}}, p, va) \stackrel{\text{def}}{=} \textit{get-bpte}(c_{\text{ISA}}, p, bpx(va)) \cdot 2^{22} + bbx(va).$$

Taking a decision. From a page table entry pte we extract the valid bit at bit position 11:

$$v(pte) \stackrel{\text{def}}{=} pte/2^{12} \bmod 2.$$

The value of this bit signals whether the page we are considering resides in the main or swap memory. We define the function $mem :: C_{\text{ISA+DS}} \times \mathbb{N} \mapsto Mem_{\text{ASM}}$ which makes this decision and creates an assembly memory components which we use in the reconstruction functions defined above in this chapter:

$$mem :: C_{\text{ISA+DS}} \times \mathbb{N} \mapsto Mem_{\text{ASM}}$$

$$mem(c_{\text{ISA+DS}}, p)(va) \stackrel{\text{def}}{=} \begin{cases} int_{\text{vars}}(c_{\text{ISA+DS}}.cpu, pma(c_{\text{ISA+DS}}.cpu, p, va \cdot 4)) & \text{if } v? \\ int_{\text{swap}}(c_{\text{ISA+DS}}.devs, sma(c_{\text{ISA+DS}}.cpu, p, va \cdot 4)) & \text{otherwise} \end{cases},$$

where $v? = v(\textit{get-pte}(c_{\text{ISA+DS}}.cpu, p, px(va \cdot 4))) = 1$.

The $\mathcal{B}$-relation

We have completed the definition of the function $vm(c_{\text{ISA+DS}}, p)$ which reconstructs a virtual machines for a process p from an ISA machine with devices. Now we are able to define a correctness criterion for user processes which we call the $\mathcal{B}$-relation.

The basic idea behind the $\mathcal{B}$-relation is to reconstruct virtual machines for every user process and check whether each of these machines match the one specified by the CVM user process component *ups*. For this we define an equality check operator for assembly machines. At first glance a definition of such operator should simply equate respective components of two assembly configurations. Here come two peculiarities. First, we can

check only those special purpose registers that are stored in process control blocks or other kernel variables. With the following function which takes an assembly register file *regs* as an argument we define the special registers concerned:

$$\textit{stored-spr}(\textit{regs}) \stackrel{\text{def}}{=} [\textit{regs}[\textit{sr}], \textit{regs}[\textit{pto}], \textit{regs}[\textit{ptl}], \textit{regs}[\textit{mode}]].$$

The second peculiarity arises when we compare memories. Since we model assembly memory as a mapping from address given as natural numbers to integer contents we deal with 2^{32} memory cells. However, not all of them are initialized and store sensible values which could be compared. As a solution we introduce a parameter *vm-size* which denotes the upper bound for memory addresses to be read and checked.

Definition 7.2 (Assembly configurations equality) Two assembly configurations c^1_{ASM} and c^2_{ASM} are compared by means of the following predicate:

$$\begin{aligned}
\textit{asm-equal}(c^1_{\text{ASM}}, c^2_{\text{ASM}}, \textit{vm-size}) \stackrel{\text{def}}{=}\ & c^1_{\text{ASM}}.\textit{dpc} = c^2_{\text{ASM}}.\textit{dpc} \\
\wedge\ & c^1_{\text{ASM}}.\textit{pc} = c^2_{\text{ASM}}.\textit{pc} \\
\wedge\ & \textit{tl}(c^1_{\text{ASM}}.\textit{gpr}) = \textit{tl}(c^2_{\text{ASM}}.\textit{gpr}) \\
\wedge\ & \textit{stored-spr}(c^1_{\text{ASM}}.\textit{spr}) = \textit{stored-spr}(c^2_{\text{ASM}}.\textit{spr}) \\
\wedge\ & \forall\ a < \textit{vm-size} :\ c^1_{\text{ASM}}.m(a/4) = c^2_{\text{ASM}}.m(a/4).
\end{aligned}$$

Isabelle: `cvm/map/B_relation.ASMcore_equality`

The only thing that remains before we can state the desired correctness relation for user processes is the right choice of the parameter *vm-size*. For each process p we should compare only those virtual memory parts that have been allocated. The size of the allocated memory measured in pages is stored in the page table length register and is expresses by the formula $\textit{get-ptl}(c_{\text{ISA}}, p) + 1$. Note that one is added since due to the fact that a process has no memory is signal by the *ptl* value -1.

Definition 7.3 (User processes relation) The $\mathcal{B}$-relation, denoted by $\mathcal{B}(\textit{ups}, c_{\text{ISA+DS}})$, is nothing but an equality of assembly machines for all user processes para-metrized with the right amount of virtual memory:

$$\begin{aligned}
\mathcal{B}(\textit{ups}, c_{\text{ISA+DS}}) \stackrel{\text{def}}{=}\ & \forall\ 0 < p < \texttt{PID_MAX} : \\
& \textit{asm-equal}(\textit{vm}(c_{\text{ISA+DS}}, p), \textit{ups}(p), (\textit{get-ptl}(c_{\text{ISA+DS}}.\textit{cpu}, p) + 1) \cdot 2^{12}).
\end{aligned}$$

Isabelle: `cvm/map/B_relation.B_relation`

7.1.4 Relation for the Kernel

The correctness relation for the abstract kernel component *ak* of a CVM configuration is the most involved one. Since only after linking the abstract kernel with the CVM implementation it can run on the target architecture we relate it to the ISA machine indirectly through a concrete kernel C0 machine c_{C0}. Concrete and abstract kernels are connected by the relation $\textit{kernel-rel}(\pi_{\text{AK}}, \textit{ak}, c_{\text{C0}})$ which has to hold during kernel executions, or $\textit{kernel-rel}_{\text{weak}}(\pi_{\text{AK}}, \textit{ak}, c_{\text{C0}})$ which has to hold during user steps. Configurations of the concrete kernel c_{C0} can be mapped to the hardware model by the C0-ISA

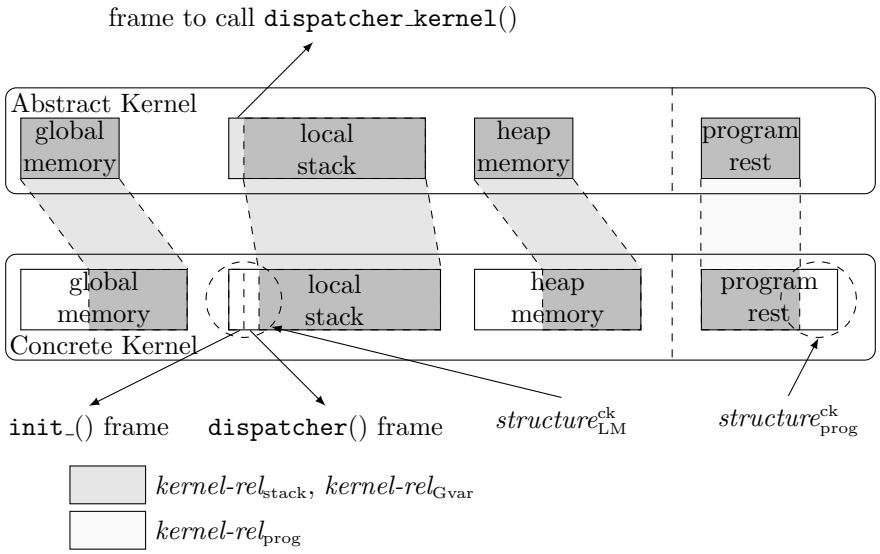

Figure 7.2: Relating memories and program rests of abstract and concrete kernels.

simulation relation. Figure 7.2 sketches the idea behind the relation for the kernel. We continue by introducing individual terms constituting both kernel relations.

Definition 7.4 (Relation for the program rest) The coupling relation for program rests of the abstract and concrete kernels is a disjunction of two terms. The first disjunct describes a special case — the abstract kernel invocation — and states that both C0 machines call the main function of the abstract kernel called `dispatcher_kernel`() with exactly the same values of parameters. The second disjunct covers all other cases of the kernel execution. It states that statements of the concrete kernel were subject to renumbering by the linker. Formally:

$$
\begin{aligned}
&\textit{kernel-rel}_{\text{prog}}(ak, c_{\text{C0}}) \stackrel{\text{def}}{=} \\
&\qquad \exists\, \textit{eca}, \textit{edata}, \textit{params}, \textit{sid}: \\
&\qquad\quad ak.\textit{prog} = ak_{\text{prog}}(\textit{eca}, \textit{edata}) \\
&\qquad \wedge \quad \textit{left-stmt}(c_{\text{C0}}.\textit{prog}) = \\
&\qquad\qquad\qquad \mathit{sCall}(\texttt{cup}, \texttt{dispatcher\_kernel}, \textit{params}, \textit{sid}) \\
&\qquad \wedge \quad \textit{reval}(\textit{link}_{\text{te}}(te_{\text{CVM}}, ak.te), c_{\text{C0}}.\textit{mem}, \textit{params}[0]) = \lfloor \mathit{nat}(\textit{eca}) \rfloor \\
&\qquad \wedge \quad \textit{reval}(\textit{link}_{\text{te}}(te_{\text{CVM}}, ak.te), c_{\text{C0}}.\textit{mem}, \textit{params}[1]) = \lfloor \mathit{nat}(\textit{edata}) \rfloor. \\
&\quad \vee \quad \textit{left-stmt}(c_{\text{C0}}.\textit{prog}) = \textit{renum}^{\text{odd}}_{\text{stmt}}(\textit{ext-upd}_{\text{stmt}}(\textit{ft}_{\text{CVM}}, ak.\textit{prog}))
\end{aligned}
$$

Isabelle: `cvm/cvm_correct/cvm_sim.kernel_relation_prog`

Recall that *left-stmt* yields the left statement of a composition statement. We elaborate why we use this function in the above definition further in this chapter when we describe implementation invariants.

While designing a correctness relation for variables of the abstract and concrete kernels one must keep in mind that there is a number of additional local frames in the

concrete kernel, in our case — one. Moreover, indices of heap variables are shifted by a constant amount of `CVM_HST_LEN`. In order to formalize these facts we introduce the following function.

Definition 7.5 (Conversion for kernel variables) For an abstract kernel g-variable g the function $\mathit{abs2conc}_{\mathrm{gvar}}$ creates the corresponding variables of the concrete kernel:

$$\mathit{abs2conc}_{\mathrm{gvar}}(g) \stackrel{\mathrm{def}}{=} \begin{cases} \mathit{gvar}_{\mathrm{gm}}(vn) & \text{if } g = \mathit{gvar}_{\mathrm{gm}}(vn) \\ \mathit{gvar}_{\mathrm{lm}}(i+1, vn) & \text{if } g = \mathit{gvar}_{\mathrm{lm}}(i, vn) \\ \mathit{gvar}_{\mathrm{hm}}(i + \texttt{CVM\_HST\_LEN}) & \text{if } g = \mathit{gvar}_{\mathrm{hm}}(i) \\ \mathit{gvar}_{\mathrm{arr}}(\mathit{abs2conc}_{\mathrm{gvar}}(v), n) & \text{if } g = \mathit{gvar}_{\mathrm{arr}}(v, n) \\ \mathit{gvar}_{\mathrm{str}}(\mathit{abs2conc}_{\mathrm{gvar}}(v), sn) & \text{if } g = \mathit{gvar}_{\mathrm{str}}(v, sn) \end{cases}.$$

Isabelle: `cvm/cvm_correct/cvm_sim.shift_local_heap_gvar`

As for the memory cell contents of the abstract and concrete kernels, the only place where they could differ is the value of a pointer variable. The reason is that the pointer targets are modeled as g-variables in the C0 small step semantics.

Definition 7.6 (Conversion for kernel memory cells) The function following $\mathit{abs2conc}_{\mathrm{cont}}$ transforms a memory cell mc of the abstract kernel to the one of the concrete kernel:

$$\mathit{abs2conc}_{\mathrm{cont}}(mc) \stackrel{\mathrm{def}}{=} \begin{cases} \mathit{ptr}(\mathit{abs2conc}_{\mathrm{gvar}}(g)) & \text{if } mc = \mathit{ptr}(g) \\ mc & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/cvm_correct/cvm_sim.shift_pointer`

With the help of these two function we are able to introduce the desired relation for kernel variables. Note that there is only one variable which is present in the abstract kernel and has no (even an) equivalent in the concrete kernel: the artificial variable `abs_kernel_res` for the result of abstract kernel computations (cf. Definition 5.2).

Definition 7.7 (Relation for kernel variables) The relation for kernel variables $\mathit{kernel\text{-}rel}_{\mathrm{Gvar}}$ compares memory configurations of the abstract and concrete kernels. It states that all abstract kernel g-variables except $\mathit{gvar}_{\mathrm{lm}}(0, \texttt{abs\_kernel\_res})$ and the correspondingly created via $\mathit{abs2conc}_{\mathrm{gvar}}$ concrete kernel variables have the same values in case they are initialized:

$$\begin{aligned} &\mathit{kernel\text{-}rel}_{\mathrm{Gvar}}(\mathit{mem}_{\mathrm{abs}}, \mathit{mem}_{\mathrm{conc}}) \stackrel{\mathrm{def}}{=} \\ &\quad \forall\, g \in \mathit{reachable}_{\mathrm{g}}(\mathit{mem}_{\mathrm{abs}}) \setminus \{\mathit{gvar}_{\mathrm{lm}}(0, \texttt{abs\_kernel\_res})\} : \\ &\qquad\qquad g \in \mathit{reachable}_{\mathrm{g}}(\mathit{mem}_{\mathrm{conc}}) \\ &\qquad \wedge \quad \mathit{initialized}_{\mathrm{g}}(\mathit{mem}_{\mathrm{conc}}, \mathit{abs2conc}_{\mathrm{gvar}}(g)) \\ &\qquad\qquad\quad = \mathit{initialized}_{\mathrm{g}}(\mathit{mem}_{\mathrm{abs}}, g) \\ &\qquad \wedge \quad \mathit{initialized}_{\mathrm{g}}(\mathit{mem}_{\mathrm{abs}}, g) \\ &\qquad\qquad \longrightarrow\ \forall i :\ \mathit{value}_{\mathrm{g}}(\mathit{mem}_{\mathrm{conc}}, \mathit{abs2conc}_{\mathrm{gvar}}(g))(i) \\ &\qquad\qquad\qquad\qquad = \mathit{abs2conc}_{\mathrm{cont}}(\mathit{value}_{\mathrm{g}}(\mathit{mem}_{\mathrm{abs}}, g)(i)). \end{aligned}$$

Isabelle: `cvm/cvm_correct/cvm_sim.shifted_memory`

It is the case that local stacks of the abstract and concrete kernel have common suffixes and differ only in their first elements. At the bottom of the abstract kernel's local stack we have an artificial memory frame (cf. Definition 5.2) while the first two elements of the concrete kernel are frames for the function `init_`() and the CVM dispatcher. The remaining parts of these stacks coincide. The following definition describes this structure formally.

Definition 7.8 (Relation for kernel local stacks) The relation for kernel local stacks stated below consists of the conjuncts which have the following meaning. First, the stack of the concrete kernel is one element longer than the one of the abstract kernel. The result variable of the abstract kernel's second frame is the artificial variable `abs_kernel_res`. The local symbol table of the first frame contains only this variable. The remaining two conjuncts claim that all other result and local variables coincide in both local stacks. Formally:

$$
\begin{aligned}
&\mathit{kernel\text{-}rel}_{\mathrm{stack}}(lm_{\mathrm{abs}}, lm_{\mathrm{conc}}) \stackrel{\mathrm{def}}{=} \\
&\qquad 2 \leq |lm_{\mathrm{abs}}| + 1 = |lm_{\mathrm{conc}}| \\
&\wedge \quad 2 \leq |lm_{\mathrm{abs}}| \;\longrightarrow\; lm_{\mathrm{abs}}[\![1]\!].\mathit{res} = \mathit{gvar}_{\mathrm{lm}}(0, \texttt{abs\_kernel\_res}) \\
&\wedge \quad lm_{\mathrm{abs}}[\![0]\!].\mathit{mfr.st} = [(\texttt{abs\_kernel\_res}, \mathit{unsgnd}_{\mathrm{T}})] \\
&\wedge \quad \forall\, i < |lm_{\mathrm{abs}}| - 2 : \\
&\qquad\qquad lm_{\mathrm{conc}}[\![i+3]\!].\mathit{res} = \mathit{abs2conc}_{\mathrm{gvar}}(lm_{\mathrm{abs}}[\![i+2]\!].\mathit{res}) \\
&\wedge \quad \forall\, i < |lm_{\mathrm{abs}}| - 1 :\; lm_{\mathrm{abs}}[\![i+1]\!].\mathit{mfr.st} = lm_{\mathrm{conc}}[\![i+2]\!].\mathit{mfr.st}.
\end{aligned}
$$

Isabelle: `cvm/cvm_correct/cvm_sim.kernel_relation_mem`

The result of an abstract kernel execution is supposed to be the process identifier of the next scheduled process. In the abstract kernel this result is written into the artificial result variable `abs_kernel_res`. In the concrete kernel the same value is written into the variable for current process identifier. The relation below bridges the values of these two variables.

Definition 7.9 (Relation for abstract kernel executions results) The relation for abstract kernel execution results holds if the abstract kernel and the concrete kernel results are equal:

$$
\begin{aligned}
\mathit{kernel\text{-}rel}_{\mathrm{result}}(ak, c_{\mathrm{C0}}) \quad \stackrel{\mathrm{def}}{=} \quad & \texttt{abs\_kernel\_res} \in ak.\mathit{mem.lm}[\![0]\!].\mathit{mfr.init} \\
\wedge \quad & \mathit{value}_{\mathrm{g}}(c_{\mathrm{C0}}.\mathit{mem}, \mathit{gvar}_{\mathrm{gm}}(\texttt{cup})) = \\
& \mathit{value}_{\mathrm{g}}(ak.\mathit{mem}, \mathit{gvar}_{\mathrm{lm}}(0, \texttt{abs\_kernel\_res})).
\end{aligned}
$$

Isabelle: `cvm/cvm_correct/cvm_sim.kernel_result_relation`

So far, we have defined individual correctness relations for such parts of the abstract and concrete kernels as program rests, variables, local memory stacks, and execution results. In the following we put them together into two predicates. One of them is our correctness notion for the kernel during its runs. The other is the correctness statement of the suspended kernel and is supposed to hold during user executions. Both relations are defined between the monolithic C0 configuration ak of the abstract kernel and the C0 configuration c_{C0} of the concrete kernel. As an additional parameter the relations take the abstract kernel program π_{AK}.

Definition 7.10 (Kernels relations) The simulation relation between the abstract and concrete kernels during kernel executions states, first of all, that type environments, function tables as well as global symbol tables of the abstract kernel configuration ak coincide with the respective components of the abstract kernel program π_{AK}. Next, the relation defines the heap symbol table of the concrete kernel to be a concatenation of the heap variables hst_{CVM} from the CVM implementation and the heap symbol table of the abstract kernel. Further, the relations for variables, local stacks, and program rest hold. Finally, whenever the abstract kernel finishes its jobs, indicated by an empty statement in the program rest, the relation for the executions results takes place. Formally:

$$\begin{aligned}
\mathit{kernel\text{-}rel}(\pi_{\text{AK}}, ak, c_{\text{C0}}) \stackrel{\text{def}}{=}\ & ak.te = \pi_{\text{AK}}.te \\
\wedge\ & ak.ft = \pi_{\text{AK}}.ft \\
\wedge\ & gst(ak.mem) = \pi_{\text{AK}}.st \\
\wedge\ & hst(c_{\text{C0}}.mem) = hst_{\text{CVM}} \circ hst(ak.mem) \\
\wedge\ & \mathit{kernel\text{-}rel}_{\text{Gvar}}(ak.mem, c_{\text{C0}}.mem) \\
\wedge\ & \mathit{kernel\text{-}rel}_{\text{stack}}(ak.mem.lm, c_{\text{C0}}.mem.lm) \\
\wedge\ & \mathit{kernel\text{-}rel}_{\text{prog}}(ak, c_{\text{C0}}) \\
\wedge\ & \mathit{is\text{-}skip}(ak.prog) \longrightarrow \mathit{kernel\text{-}rel}_{\text{result}}(ak, c_{\text{C0}}).
\end{aligned}$$

The stack of local frames and the program rest are not maintained during user executions. Therefore, the simulation relation between the abstract and concrete kernels during user executions lacks three last conjuncts of the relation above:

$$\begin{aligned}
\mathit{kernel\text{-}rel}_{\text{weak}}(\pi_{\text{AK}}, ak, c_{\text{C0}}) \stackrel{\text{def}}{=}\ & ak.te = \pi_{\text{AK}}.te \\
\wedge\ & ak.ft = \pi_{\text{AK}}.ft \\
\wedge\ & gst(ak.mem) = \pi_{\text{AK}}.st \\
\wedge\ & hst(c_{\text{C0}}.mem) = hst_{\text{CVM}} \circ hst(ak.mem) \\
\wedge\ & \mathit{kernel\text{-}rel}_{\text{Gvar}}(ak.mem, c_{\text{C0}}.mem).
\end{aligned}$$

Isabelle: `cvm/cvm_correct/cvm_sim.kernel_relation`
`cvm/cvm_correct/cvm_sim.weak_kernel_relation`

7.1.5 Combining Relations for Components

Now we can formally define CVM correctness relations $\mathit{cvm\text{-}sim}_{\text{kernel}}$ and $\mathit{cvm\text{-}sim}_{\text{weak}}$ which we have mentioned in the beginning of this section. The relation which has to hold during kernel execution is a conjunction of the relation for the kernel, for devices, for user processes, and the equality between the status register value retrieved from the memory of VAMP ISA and the corresponding CVM component:

$$\begin{aligned}
\mathit{cvm\text{-}sim}_{\text{kernel}}(\pi_{\text{AK}}, c_{\text{CVM}}, c_{\text{C0}}, c_{\text{ISA+DS}}) \stackrel{\text{def}}{=}\ & \mathit{kernel\text{-}rel}(c_{\text{CVM}}.ak, c_{\text{C0}}, \pi_{\text{AK}}) \\
\wedge\ & \mathit{dev\text{-}rel}(c_{\text{CVM}}, c_{\text{ISA+DS}}) \\
\wedge\ & \mathcal{B}(c_{\text{CVM}}.ups, c_{\text{ISA+DS}}) \\
\wedge\ & val_{sr}(c_{\text{ISA+DS}}) = c_{\text{CVM}}.sr.
\end{aligned}$$

As for the CVM correctness relation which has to hold during user executions, it is the relation above extended with one term. The value of the variable for current process identifier obtained from the memory of the physical combined system is equal to the given value *pid*:

$$\begin{aligned} \textit{cvm-sim}_{\text{weak}}(\pi_{\text{AK}}, c_{\text{CVM}}, c_{\text{C0}}, c_{\text{ISA+DS}}, \textit{pid}) \stackrel{\text{def}}{=} & \quad \textit{kernel-rel}_{\text{weak}}(c_{\text{CVM}}.\textit{ak}, c_{\text{C0}}, \pi_{\text{AK}}) \\ \wedge & \quad \textit{dev-rel}(c_{\text{CVM}}, c_{\text{ISA+DS}}) \\ \wedge & \quad \mathcal{B}(c_{\text{CVM}}.\textit{ups}, c_{\text{ISA+DS}}) \\ \wedge & \quad \textit{val}_{sr}(c_{\text{ISA+DS}}) = c_{\text{CVM}}.sr \\ \wedge & \quad \textit{val}_{cup}(c_{\text{ISA+DS}}) = \textit{pid}. \end{aligned}$$

7.1.6 Implementation Invariants

It turns out that in order to be able to prove that the previously defined CVM correctness criteria hold throughout CVM executions a number of invariants over the concrete kernel's C0 machines as well as the underlying ISA machine with device has to hold. We call such properties *implementation invariants*. In the following we define them formally.

All implementation invariants are collected in the predicate

$$\textit{impl-inv} :: \Pi_{\text{C0}} \times C_{\text{CVM}} \times C_{\text{C0}} \times C_{\text{ISA+DS}} \mapsto \mathbb{B}$$

which, essentially, speaks about the C0 machine c_{C0} of the concrete kernel and the physical combined system configuration $c_{\text{ISA+DS}}$. Additionally, the predicate takes the abstract kernel program π_{AK} and the CVM state c_{CVM} as arguments. Note that the CVM configuration is passed to the predicate only to distinguish one of the three execution cases: a waiting for an interrupt state, a kernel step, or a user step. Formally, these cases are distinguished in completely the same way as in the definition of the CVM correctness relation *cvm-sim* which has been introduced in the beginning of this section. Ultimately, the formal definition of implementation invariants is as follows.

$$\textit{impl-inv}(\pi_{\text{AK}}, c_{\text{CVM}}, c_{\text{C0}}, c_{\text{ISA+DS}}) \stackrel{\text{def}}{=} \begin{cases} \textit{impl-inv}_{\text{wait}}(\pi_{\text{AK}}, c_{\text{C0}}, c_{\text{ISA+DS}}.\textit{cpu}) & \text{if} \quad c_{\text{CVM}}.\textit{cup} = \bot \\ & \quad \wedge \ \textit{is-wait-state}(c_{\text{CVM}}) \\ \textit{impl-inv}_{\text{kernel}}(\pi_{\text{AK}}, c_{\text{C0}}, c_{\text{ISA+DS}}.\textit{cpu}) & \text{if} \quad c_{\text{CVM}}.\textit{cup} = \bot \\ & \quad \wedge \ \neg \textit{is-wait-state}(c_{\text{CVM}}) \\ \textit{impl-inv}_{\text{user}}(\pi_{\text{AK}}, c_{\text{C0}}, c_{\text{ISA+DS}}.\textit{cpu}, \textit{pid}) & \text{if} \quad c_{\text{CVM}}.\textit{cup} = \lfloor \textit{pid} \rfloor \end{cases}.$$

We continue with introduction of building blocks constituting definitions of each case.

Page-Fault Handler Invariants

The page-fault handler related code constitutes a significant part of the CVM implementation. It was verified separately using a number of different techniques compared to those used in this thesis, in particular a semantics stack [4, 107]. However, the handler is heavily called from the remaining CVM implementation for treating page-faults and address translation in primitives. Correctness of the page-fault handler is expressed by a number of invariants which hold during its executions and must not be destroyed by the CVM functions. In order to guarantee that we make the page-fault handler invariant a

part of the CVM implementation invariants. For the definitions of the mentioned below predicates the reader should consult the thesis of Starostin [105].

The first of the page-fault handler related invariants is defined between a page-fault handler abstract state $c_{\text{PFH}} :: C_{\text{PFH}}$ and a memory configuration mc of the concrete kernel with respect to the kernel's type environment te. It states that the memory configuration encodes the abstract PFH state, and that the latter is valid. Formally this invariant is defined by the predicate

$$\textit{pfh-inv}(c_{\text{PFH}}, te, mc).$$

The next invariant is called the zero filled page condition. Denoted by

$$\textit{zfp-cond}(c_{\text{ISA}}),$$

it states that a page filled with zeros resides at the address `ZFP` in the memory of the ISA machine c_{ISA}.

The page-fault handler was verified in the Hoare logic environment and its correctness results were transferred to the C0 small step semantics level using Verisoft's semantics stack. The stack imposes additional validity criteria for C0 programs, denoted by the predicate

$$\textit{hoare-C0-validity}(te, mc).$$

We update our definition of C0 validity by adding this predicate as follows:

$$\begin{array}{rl} & c_{\text{C0}} \in C0'\surd(te, ft) \\ \wedge & \textit{hoare-C0-validity}(te, c_{\text{C0}}.\textit{mem}) \\ \longrightarrow & c_{\text{C0}} \in C0''\surd(te, ft). \end{array}$$

ISA Invariants

This group of invariants states properties of the target VAMP ISA configuration which has to be respected during CVM runs. The invariants mainly speak about the translated kernel code residing in the ISA memory and about the values of some special registers.

We read a list of n integer values from the memory of a VAMP ISA machine c_{ISA} starting at address a by means of the function

$$\textit{get-data}_{\text{ISA}}(c_{\text{ISA}}, a, n) \stackrel{\text{def}}{=} [\textit{int}_{\text{vars}}(c_{\text{ISA}}, a), \ldots, \textit{int}_{\text{vars}}(c_{\text{ISA}}, a + 4 \cdot (n-1))].$$

We convert an assembly program given by a list of instructions π_{ASM} to a list of integers by means of the function

$$\textit{asm}_{\pi}\textit{-to-ints}(\pi_{\text{ASM}}) \stackrel{\text{def}}{=} il,$$

such that

$$il[i] = \textit{instr-to-int}(\pi_{\text{ASM}}[i]).$$

Definition 7.11 (Concrete kernel code invariant) The following invariant ensure that two facts take place. First, the instruction stored in the ISA memory at address zero must be a jump instruction to the beginning of the kernel code (and the next instruction is `nop`). Note, that the immidiate constant of the jump instruction must be 4 bytes less than the relative jump destination address, because the semantics of the

jump instruction adds the constant to the program counter which is greater by 4 than the delayed program counter, i.e., the address of the current instruction. Second, the predicate below ensures that the concrete kernel code translated by the C0 compiler resides in the ISA memory starting from the address `PROGBASE`:

$$\begin{aligned}
&\mathit{code\text{-}inv}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \\
&\qquad \mathit{get\text{-}data}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, 0, 2) = \mathit{asm}_{\pi}\text{-}\mathit{to\text{-}ints}([\mathtt{j}(\mathtt{PROGBASE} - 4), \mathtt{nop}]) \\
\wedge \quad &\mathit{get\text{-}data}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, \mathtt{PROGBASE}, \mathit{csize}_{\mathrm{prog}}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{gst}_{\mathrm{CK}}(\pi_{\mathrm{AK}}))) \\
&\qquad = \mathit{asm}_{\pi}\text{-}\mathit{to\text{-}ints}(\mathit{code}_{\mathrm{prog}}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{gst}_{\mathrm{CK}}(\pi_{\mathrm{AK}}))).
\end{aligned}$$

Isabelle: `cvm/config/isa_config.code_invariant_isa`

The next invariant is used to describe the waiting for interrupts case. The case is modeled by an infinite loop consisting of two instructions: $\mathtt{j}(-4)$ and `nop`.

Definition 7.12 (Program counters invariant for wait case) The predicate below states that the infinite loop program is located in the memory of an ISA configuration c_{ISA} starting from some address ad and that the program counters point inside this program:

$$\begin{aligned}
\mathit{pc\text{-}inv}_{\mathrm{wait}}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \exists\, ad: \quad &\mathit{get\text{-}data}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, ad, 2) = \mathit{asm}_{\pi}\text{-}\mathit{to\text{-}ints}([\mathtt{j}(-4), \mathtt{nop}]) \\
\wedge \quad &\langle c_{\mathrm{ISA}}.\mathit{dpc}\rangle = ad \ \wedge\ \langle c_{\mathrm{ISA}}.\mathit{pc}\rangle = ad + 4 \\
\vee \quad &\langle c_{\mathrm{ISA}}.\mathit{dpc}\rangle = ad + 4 \ \wedge\ \langle c_{\mathrm{ISA}}.\mathit{pc}\rangle = ad.
\end{aligned}$$

Isabelle: `cvm/cvm_correct/cvm_sim.pc_invariant_isa`
The last ISA invariant is supposed to hold during user executions.

Definition 7.13 (User execution invariant) The following predicate guarantees that no user process harms the system by one or several of the following facts: (i) rescheduling processes by writing the current process identifier, (ii) masking interrupts by writing the status register, or (iii) changing the *pto* or *ptl* registers. Formally, the predicate below states that the process identifier variable, the status register as well as *pto* and *ptl* registers remain unchanged during user executions:

$$\begin{aligned}
\mathit{user\text{-}inv}(c_{\mathrm{ISA}}, \mathit{pid}) \stackrel{\mathrm{def}}{=} \quad &\mathit{pid} = \mathit{nat}_{\mathrm{vars}}(c_{\mathrm{ISA}}, \mathit{ad}_{\mathit{cup}}) \\
\wedge \quad &c_{\mathrm{ISA}}.\mathit{spr}(\mathit{sr}) = \mathit{read\text{-}isa}(c_{\mathrm{ISA}}.\mathit{Mem}_{\mathrm{ISA}}, \mathit{ad}_{\mathit{sr}}) \\
\wedge \quad &c_{\mathrm{ISA}}.\mathit{spr}(\mathit{pto}) = \mathit{read\text{-}isa}(c_{\mathrm{ISA}}.\mathit{Mem}_{\mathrm{ISA}}, \mathit{ad}^{\mathit{pid}}_{\mathit{PTO}_{\mathrm{EF}}}) \\
\wedge \quad &c_{\mathrm{ISA}}.\mathit{spr}(\mathit{ptl}) = \mathit{read\text{-}isa}(c_{\mathrm{ISA}}.\mathit{Mem}_{\mathrm{ISA}}, \mathit{ad}^{\mathit{pid}}_{\mathit{PTL}_{\mathrm{EF}}}).
\end{aligned}$$

Isabelle: `cvm/additional/isa_defs.user_exec_invariant`

Weak Ministack for the Kernel

User and kernel computations interleave within CVM model executions. The correctness of the kernel expressed by the C0-ISA simulation relation to which we also refer as the ministack relation has to hold after an execution of every statement of the kernel. However, a C0 small-step semantics configuration encoding the kernel and consistent to

the underlying hardware cannot be simply constructed from an ISA machine configuration after a user execution. There are two reasons for that: (i) there is no information about C0 types in the memory model of an ISA machine, and (ii) there is no one-to-one mapping between the heap of an ISA machine and the heap memory frame of a C0 configuration. In order to be able to resume a C0 kernel computation after a user execution users have to respect the constraint that they do not affect kernel code and kernel data, including the mapping between heaps. We call this constraint *weak C0 ministack* and define it formally below.

The parts of a C0 configuration encoding the kernel that is can be preserved unchanged during user computations are the global memory frame and the heap memory frame. As for the local memory and program rest, even if they are not empty before the switch to the user execution, the kernel starts next time with new stack and program rest. We call a C0 configuration in which only these two components are defined and all other left empty a *weak C0 configuration*.

We define the set of *C0 weakly valid* configurations:

$$c_{\mathrm{C0}} \in \mathit{C0}_{\mathrm{weak}}\surd(\mathit{te}, \mathit{ft}).$$

It differs from the ordinary set of valid C0 programs only in that it lacks all terms concerning local stack and a program rest.

A weak C0 configuration is related to an ISA state by means of the *weak C0 consistency relation*. This relation reuses sub-relations of the normal C0 consistency and a one additional term which we define first.

Definition 7.14 (Kernel heap consistency) The kernel heap consistency relation states that the kernel variable `kheap` stores the first unused address on the heap:

$$\mathit{consis}_{\mathrm{kheap}}(c_{\mathrm{C0}}, \mathit{kheap}) \stackrel{\mathrm{def}}{=} \mathit{kheap} = \mathtt{ABASE}_{\mathrm{hm}} + \mathit{asize}_{\mathrm{heap}}(\mathit{hst}(c_{\mathrm{C0}}.\mathit{mem})).$$

Isabelle: `cvm/config/weak_conditions.kheap_consistent`

It might be seen as a part of the register consistency $\mathit{consis}_{\mathrm{r}}$.

Definition 7.15 (Weak C0 consistency) The weak C0 consistency is defined by the predicate

$$\mathit{consis}_{\mathrm{weak}} :: \mathit{Tenv} \times \mathit{Functable} \times C_{\mathrm{C0}} \times \mathit{Alloc} \times C_{\mathrm{ASM}} \times \mathbb{N} \mapsto \mathbb{B},$$

which is a conjunction of the (i) code consistency, which guarantees that users do not modify the kernel code, (ii) value and pointer consistency, which guarantees that users do not modify the kernel data, (iii) the allocation consistency and the kernel heap consistency, which defines the mapping between heaps of an ISA machine and C0 configuration. Formally:

$$\begin{aligned}
\mathit{consis}_{\mathrm{weak}}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}, \mathit{kheap}) \stackrel{\mathrm{def}}{=}\ & \mathit{consis}_{\mathrm{code}}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{v}}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{p}}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, c_{\mathrm{ASM}}) \\
\wedge\ & \mathit{consis}_{\mathrm{alloc}}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, \mathit{alloc}) \\
\wedge\ & \mathit{consis}_{\mathrm{kheap}}(c_{\mathrm{C0}}, \mathit{kheap}).
\end{aligned}$$

Isabelle: `cvm/config/weak_conditions.weak_consistent`

The weak C0-ISA simulation relation, or the weak ministack, is a composition of the weak C0 consistency relation together with the simulation relation between VAMP assembly and ISA.

Definition 7.16 (Weak ministack) The weak ministack predicate has the following signature:

$$\mathit{C0\text{-}sim\text{-}isa}_{\mathrm{weak}} :: \mathit{Tenv} \times \mathit{Functable} \times C_{\mathrm{C0}} \times C_{\mathrm{ISA}} \mapsto \mathbb{B},$$

and is formally defined as follows:

$$\begin{array}{lcl} \mathit{C0\text{-}sim\text{-}isa}_{\mathrm{weak}}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ISA}}) & \stackrel{\mathrm{def}}{=} & \exists\, c_{\mathrm{ASM}}, \mathit{alloc} : \\ & & \mathit{asm}\surd(c_{\mathrm{ASM}}) \\ & \wedge & \mathit{consis}_{\mathrm{weak}}(te, ft, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}}, \\ & & \quad \langle \mathit{read\text{-}isa}(c_{\mathrm{ISA}}.\mathit{Mem}_{\mathrm{ISA}}, \mathit{ad}_{\mathit{kheap}})\rangle) \\ & \wedge & \mathit{isa\text{-}sim\text{-}asm}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}). \end{array}$$

Isabelle: `cvm/config/weak_conditions.weak_sim_C0_isa`

Kernel Structure Invariants

It follows from the CVM implementation that at the moment when the concrete kernel calls the dispatcher of the abstract kernel there are some remaining statement in the program rest of the concrete kernel that have to be executed afterwards. These statement constitute the code in the CVM dispatcher located after the call to the abstract kernel dispatcher: a case distinction on the identifier of the next scheduled process and invocation either of `cvm_start`() or `cvm_wait`() depending on the result. Let this part of the code be defined by the constant `disp-end-code`.

Definition 7.17 (Concrete kernel program rest invariant) The concrete kernel program rest invariant states that the program rest is a composition of two parts: the one described by $\mathit{kernel\text{-}rel}_{\mathrm{prog}}$ and the constant `disp-end-code`:

$$\begin{array}{lcl} \mathit{structure}^{\mathrm{ck}}_{\mathrm{prog}}(c_{\mathrm{C0}}) & \stackrel{\mathrm{def}}{=} & \mathit{is\text{-}comp}(c_{\mathrm{C0}}.\mathit{prog}) \\ & \wedge & \mathit{right\text{-}stmt}(c_{\mathrm{C0}}.\mathit{prog}) = \texttt{disp-end-code}. \end{array}$$

Isabelle: `cvm/cvm_correct/cvm_sim.concr_kernel_prog_structure`

As mentioned before the two topmost local memory frames of the concrete kernel are not present in the local stack of the abstract kernel.

Definition 7.18 (Concrete kernel local stack invariant) The following predicate formally states the presence of the memory frame of the CVM function `init_`() and the CVM dispatcher frame in the memory configuration of the concrete kernel:

$$\begin{array}{lcl} \mathit{structure}^{\mathrm{ck}}_{\mathrm{LM}}(c_{\mathrm{C0}}) & \stackrel{\mathrm{def}}{=} & 2 \leq |c_{\mathrm{C0}}.\mathit{mem.lm}| \\ & \wedge & 2 < |c_{\mathrm{C0}}.\mathit{mem.lm}| \longrightarrow \\ & & \quad c_{\mathrm{C0}}.\mathit{mem.lm}[\![2]\!].\mathit{res} = \mathit{gvar}_{\mathrm{gm}}(\texttt{cup}) \\ & \wedge & c_{\mathrm{C0}}.\mathit{mem.lm}[\![0]\!].\mathit{mfr.st} = \mathit{st}_{\mathrm{fun}}(\mathit{init\text{-}proc}) \\ & \wedge & c_{\mathrm{C0}}.\mathit{mem.lm}[\![1]\!].\mathit{mfr.st} = \mathit{st}_{\mathrm{fun}}(\mathit{dispatcher\text{-}proc}). \end{array}$$

Isabelle: `cvm/cvm_correct/cvm_sim.concr_kernel_lmstack_structure`

Definition 7.19 (Concrete kernel global memory invariant) First, the concrete kernel global memory invariant requires the names of global variables in the concrete kernel to be the same as in the abstract kernel program. Second, global variables of the abstract kernel must be initialized.

$$\begin{aligned} \mathit{structure}^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \quad &\stackrel{\mathrm{def}}{=} \quad \mathit{gst}(c_{\mathrm{C0}}.\mathit{mem}) = \mathit{gst}_{\mathrm{CK}}(\pi_{\mathrm{AK}}) \\ &\wedge \quad \mathit{vns}(\mathit{gst}_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \subseteq c_{\mathrm{C0}}.\mathit{mem.gm.init}. \end{aligned}$$

Isabelle: `cvm/config/software_conditions.gm_structure`

Definition 7.20 (Concrete kernel heap memory invariant) As for the structure of a concrete kernel's heap, we demand the heap symbol table of the concrete kernel to begin with the heap symbol table $\mathit{hst}_{\mathrm{CVM}}$ of the CVM implementation:

$$\mathit{structure}^{\mathrm{ck}}_{\mathrm{HM}}(c_{\mathrm{C0}}) \stackrel{\mathrm{def}}{=} \exists\, \mathit{st}:\ \mathit{hst}(c_{\mathrm{C0}}.\mathit{mem}) = \mathit{hst}_{\mathrm{CVM}} \circ \mathit{st}.$$

Isabelle: `cvm/config/software_conditions.hm_structure`

Putting Implementation Invariants Together

Having individual implementation invariants defined we can use them as building blocks for defining overall implementation invariants which are supposed to hold in each of the three cases: kernel execution, user execution, and waiting for interrupts states.

The implementation invariants $\mathit{impl\text{-}inv}_{\mathrm{kernel}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA}})$ which hold during kernel executions include (i) the validity of the ISA machine, ISA invariants, and the zero filled page condition, (ii) a requirement to the ISA machine to operate in system mode, (iii) the validity of the concrete kernel C0 configuration, (iv) all previously defined structure invariants of the concrete kernel, (v) the page-fault handler invariants, and (vi) the C0-ISA simulation relation between the concrete kernel C0 state and the ISA machine. Formally:

$$\begin{aligned} \mathit{impl\text{-}inv}_{\mathrm{kernel}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA}}) \quad &\stackrel{\mathrm{def}}{=} \quad \mathit{isa}\surd(c_{\mathrm{ISA}}) \\ &\wedge \quad \mathit{code\text{-}inv}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}}) \\ &\wedge \quad \mathit{zfp\text{-}cond}(c_{\mathrm{ISA}}) \\ &\wedge \quad \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\ &\wedge \quad c_{\mathrm{C0}} \in C0''\surd(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\ &\wedge \quad \mathit{structure}^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\ &\wedge \quad \mathit{structure}^{\mathrm{ck}}_{\mathrm{HM}}(c_{\mathrm{C0}}) \\ &\wedge \quad \mathit{structure}^{\mathrm{ck}}_{\mathrm{LM}}(c_{\mathrm{C0}}) \\ &\wedge \quad \mathit{structure}^{\mathrm{ck}}_{\mathrm{prog}}(c_{\mathrm{C0}}) \\ &\wedge \quad \exists\, c_{\mathrm{PFH}}:\ \mathit{pfh\text{-}inv}(c_{\mathrm{PFH}}, \mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\mathit{mem}) \\ &\wedge \quad \mathit{C0\text{-}sim\text{-}isa}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}, c_{\mathrm{ISA}}). \end{aligned}$$

The implementation invariant $\mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA}}, \mathit{pid})$ which holds during user executions consists of (i) the validity of the ISA machine, ISA invariants, and the zero

filled page condition, (ii) a requirement to the ISA machine to operate in user mode, (iii) the user ISA invariant, (iv) a statement that *pid* corresponds to some user process, (v) weak validity of the C0 configuration, (vi) the concrete kernel invariants for global and heap memory, (vii) the page-fault handler invariants, (viii) a requirement for the process *pid* to have some virtual memory, and (ix) the weak C0-ISA simulation relation. Formally:

$$
\begin{aligned}
\mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA}}, \mathit{pid}) \stackrel{\mathrm{def}}{=}\ & \mathit{isa}\surd(c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{code\text{-}inv}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{zfp\text{-}cond}(c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{is\text{-}user\text{-}mode}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{user\text{-}inv}(c_{\mathrm{ISA}}, \mathit{pid}) \\
\wedge\ & 0 < \mathit{pid} \wedge \mathit{pid} < \texttt{PID\_MAX} \\
\wedge\ & c_{\mathrm{C0}} \in \mathit{C0}_{\mathrm{weak}}\surd(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\
\wedge\ & \mathit{structure}^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge\ & \mathit{structure}^{\mathrm{ck}}_{\mathrm{HM}}(c_{\mathrm{C0}}) \\
\wedge\ & \exists\, c_{\mathrm{PFH}} : \\
& \qquad \mathit{pfh\text{-}inv}(c_{\mathrm{PFH}}, \mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\mathit{mem}) \\
& \quad \wedge\ \ 0 \leq c_{\mathrm{PFH}}.\mathit{pcb}[\mathit{pid}].\mathit{ptl} \\
\wedge\ & \mathit{C0\text{-}sim\text{-}isa}_{\mathrm{weak}}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}, c_{\mathrm{ISA}}).
\end{aligned}
$$

The implementation invariants $\mathit{impl\text{-}inv}_{\mathrm{wait}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA}})$ which hold during the waiting for interrupts state are (i) the validity of the ISA machine, ISA invariants, and the zero filled page condition, (ii) a requirement to the ISA machine to operate in system mode, (iii) a statement that the kernel variable `sr` stores the same value which resides in the *sr* register, (iv) the invariant for program counters, (v) the weak validity of the C0 configuration, (vi) the concrete kernel invariants for global and heap memory, (vii) the page-fault handler invariants, and (viii) the weak C0-ISA simulation relation. Formally:

$$
\begin{aligned}
\mathit{impl\text{-}inv}_{\mathrm{wait}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=}\ & \mathit{isa}\surd(c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{code\text{-}inv}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{zfp\text{-}cond}(c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{is\text{-}sys\text{-}mode}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\
\wedge\ & c_{\mathrm{ISA}}.\mathit{spr}(\mathit{sr}) = \mathit{read\text{-}isa}(c_{\mathrm{ISA}}.m, \mathit{ad}_{\mathit{sr}}) \\
\wedge\ & \mathit{pc\text{-}inv}_{\mathrm{wait}}(c_{\mathrm{ISA}}) \\
\wedge\ & c_{\mathrm{C0}} \in \mathit{C0}_{\mathrm{weak}}\surd(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\
\wedge\ & \mathit{structure}^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge\ & \mathit{structure}^{\mathrm{ck}}_{\mathrm{HM}}(c_{\mathrm{C0}}) \\
\wedge\ & \exists\, c_{\mathrm{PFH}} :\ \mathit{pfh\text{-}inv}(c_{\mathrm{PFH}}, \mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\mathit{mem}) \\
\wedge\ & \mathit{C0\text{-}sim\text{-}isa}_{\mathrm{weak}}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}, c_{\mathrm{ISA}}).
\end{aligned}
$$

7.2 CVM Correctness Theorem

CVM top-level correctness is stated as a simulation theorem between VAMP ISA with devices and the CVM model. Before we can formulate it we have to discuss a number of assumptions

Initial ISA configuration. We start from the initial ISA configuration which is the first valid configuration after reset. We define this configuration by means of the predicate *is-init-isa*$(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}})$. It claims initial values of program counters and special purpose registers. The predicate imposes valid size requirements to the register files and memory. Additionally, it claims through the conjunct *code-inv*$(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}})$ that the concrete kernel code is placed in the memory.

$$\begin{aligned} \textit{is-init-isa}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=}\ & c_{\mathrm{ISA}}.\textit{dpc} = \textit{bin}(0) \\ \wedge\ & c_{\mathrm{ISA}}.\textit{pc} = \textit{bin}(4) \\ \wedge\ & c_{\mathrm{ISA}}.\textit{spr} = \textit{SPR}_{\mathrm{init}} \\ \wedge\ & \textit{Regf}_{\mathrm{ISA}}\surd(c_{\mathrm{ISA}}.\textit{gpr}) \\ \wedge\ & \textit{Mem}_{\mathrm{ISA}}\surd(c_{\mathrm{ISA}}.m) \\ \wedge\ & \textit{code-inv}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}}). \end{aligned}$$

Here the initial value of the special purpose register file is defined as follows:

$$\langle \textit{SPR}_{\mathrm{init}}(r) \rangle \stackrel{\mathrm{def}}{=} \begin{cases} 1 & \text{if } r = \textit{eca} \\ 0 & \text{otherwise} \end{cases}.$$

This formula expresses the fact that all special registers are initialized with zero value except for the register *eca*. The value of the latter means that only the reset exception is raised.

Validity of the swap hard disk. We connect the ISA machine in the initial configuration to a valid hard disk to store swap data on it. We define the predicate *is-HD*$\surd(c_{\mathrm{DS}})$ over a devices system which states the following facts: (i) the device in position `SWAP_DID` of the devices system is a hard disk, (ii) this hard disk is in idle state and does not produce a pending interrupt, (iii) the size of the hard disk is sufficient, (iv) the buffer size is fixed, and (v) the buffer pointer is zero:

$$\begin{aligned} \textit{is-HD}\surd(c_{\mathrm{DS}}) \stackrel{\mathrm{def}}{=}\ & \exists\, c_{\mathrm{HD}} : c_{\mathrm{DS}}(\texttt{SWAP\_DID}) = \textit{dev-hd}(c_{\mathrm{HD}}) \\ & \wedge\ c_{\mathrm{HD}}.\textit{cs} = \texttt{HD\_IDLE} \\ & \wedge\ \neg \textit{is-intr-hd}(c_{\mathrm{HD}}) \\ & \wedge\ 8 \cdot (150 + 1152 \cdot 2^{10}) \leq c_{\mathrm{HD}}.s < 2^{28} \\ & \wedge\ |c_{\mathrm{HD}}.\textit{sm}| = c_{\mathrm{HD}}.s \cdot \texttt{WORDS\_PER\_SECTOR} \\ & \wedge\ |c_{\mathrm{HD}}.\textit{buf}| = \texttt{WORDS\_PER\_SECTOR} \\ & \wedge\ c_{\mathrm{HD}}.\textit{bp} = 0. \end{aligned}$$

Memory map. We introduce values for constants that bound sizes of abstract kernel heap and stack. The heap frame is bounded by the constant

$$\texttt{HEAP-SIZE}_{\mathrm{AK}} \stackrel{\mathrm{def}}{=} 2^{24},$$

while the local stack is bounded by the constant

$$\texttt{STACK-SIZE}_{\mathrm{AK}} \stackrel{\mathrm{def}}{=} 2^{21}.$$

There is no reasonable way to bound the heap frame statically: we will state such requirement in the inductive step of the theorem directly. As for the local stack, in order to claim that executions of the abstract kernel respect its boundary we define the inductive set *SE*. It estimates the size of the local stack needed to execute a particular function. First, we define the function

$$\mathit{fun\text{-}names} :: \mathit{Stmt} \mapsto 2^{\mathbb{S}},$$

which traverses the statement tree and collects all function names which are called from the given statement. This function is defined by induction on the statement's structure. For compositional, conditional and loop statement we go deeper to sub-statements:

$$\begin{aligned} \mathit{fun\text{-}names}(\mathsf{comp}(s_1, s_2)) &= \mathit{fun\text{-}names}(s_1) \cup \mathit{fun\text{-}names}(s_2), \\ \mathit{fun\text{-}names}(\mathsf{ifte}(e, s_1, s_2, \mathit{sid})) &= \mathit{fun\text{-}names}(s_1) \cup \mathit{fun\text{-}names}(s_2), \\ \mathit{fun\text{-}names}(\mathsf{loop}(e, s, \mathit{sid})) &= \mathit{fun\text{-}names}(s). \end{aligned}$$

For the function call we extract the function's name:

$$\mathit{fun\text{-}names}(\mathsf{sCall}(\mathit{vn}, \mathit{fn}, \mathit{params}, \mathit{sid})) = \{\mathit{fn}\}.$$

For all other statement *fun-names* returns the empty set. Note that for external calls and extended calls the function also yields an empty set. This respects the C0 small-step semantics: execution of these calls does not create additional frames.

Having this, we can define the desired set.

Definition 7.21 (Local stack boundary) Let *ft* be a function table, let *fn* be a function's name, and let *sz* be a natural number. The set

$$SE :: 2^{\mathit{Functable} \times \mathbb{S} \times \mathbb{N}}$$

collects all triples which consist of a function table, a function's name and a natural number with the following property. For all elements $(\mathit{ft}, \mathit{fn}, \mathit{sz}) \in SE$ it holds that starting from a call of the function *fn* the execution with respect to the function table *ft* consumes not more than *sz* bytes for the stack. Formally:

$$\begin{aligned} & (\mathit{fn}, \mathit{proc}) \in \{\mathit{ft}\} \\ \wedge \quad & \forall\, \mathit{FN} \in \mathit{fun\text{-}names}(\mathit{proc.body}) :\ (\mathit{ft}, \mathit{FN}, \mathit{sz}') \in SE \\ \wedge \quad & \mathit{sz}' + \mathit{asize}_{\mathrm{st}}(\mathit{st}_{\mathrm{fun}}(\mathit{proc})) \leq \mathit{sz} \\ \longrightarrow \quad & (\mathit{ft}, \mathit{fn}, \mathit{sz}) \in SE. \end{aligned}$$

Isabelle: `C0compilercodegen/MoreC0compiler.max_stack_size`

This requirement is added to the predicate of the abstract kernel properties:

$$\begin{aligned} \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \stackrel{\mathrm{def}}{=} & \ \cdots \\ \wedge & \ (\pi_{\mathrm{AK}}.\mathit{ft}, \texttt{dispatcher\_kernel}, \texttt{STACK-SIZE}_{\mathrm{AK}}) \in SE. \end{aligned}$$

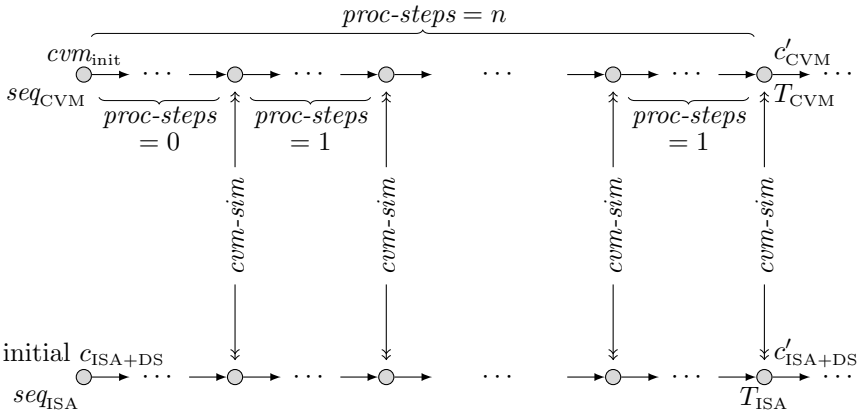

Figure 7.3: CVM simulation theorem.

Devices typing. The last definition we have to introduce before we can actually state the CVM correctness theorem is of technical nature. Since our devices system is modeled as a mapping from device identifiers to generalized device configurations we need to be sure that devices do not change their types. For this purpose we will use the relation $\overset{\text{type}}{\sim}$ which holds only for pairs of devices of the same type:

$$\begin{aligned} \textit{dev-hd}(c_{\text{HD}}) &\overset{\text{type}}{\sim} \textit{dev-hd}(c'_{\text{HD}}) \\ \textit{dev-timer}(c_{\text{TIMER}}) &\overset{\text{type}}{\sim} \textit{dev-timer}(c'_{\text{TIMER}}) \\ &\dots \\ \textit{idle-dev} &\overset{\text{type}}{\sim} \textit{idle-dev}. \end{aligned}$$

Finally we have all necessary definitions and can formulate the CVM correctness theorem. In Figure 7.3 we sketch the idea behind the theorem graphically.

Theorem 7.22 (CVM correctness theorem) Assume that (i) the abstract kernel properties hold for π_{AK}, (ii) the processor component $c_{\text{ISA+DS}}.cpu$ of ISA with devices is in its initial state, (iii) the swap hard disk properties hold for the devices system $c_{\text{ISA+DS}}.devs$, (iv) the ISA execution sequence seq_{ISA} is valid, then there exists a valid CVM execution sequence seq_{CVM} and for any finite number n of kernel steps bounded by some N there exists a number of CVM model steps T_{CVM}, such that by executing this number of steps starting from initial CVM configuration the CVM model transits to a non-error state c'_{CVM} with no heap boundaries violation. Moreover, there exists a number of ISA with devices steps T_{ISA} during which the ISA machine transits to a resulting state $c'_{\text{ISA+DS}}$ and a corresponding configuration c'_{C0} of the concrete kernel, such that (i) the devices typing and the swap hard disk properties hold for the updated devices system, (ii) the implementation invariants hold, and (iii) the CVM simulation

relation holds. Formally:

$$
\begin{array}{lll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) & \\
\wedge & \mathit{is\text{-}init\text{-}isa}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
\wedge & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
\wedge & \mathit{seq}\surd(\mathit{seq}_{\mathrm{ISA}}, c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
\longrightarrow & (\exists\, \mathit{seq}_{\mathrm{CVM}} :\ \forall\, n \leq N :\ \exists\, T_{\mathrm{CVM}} : & \\
 & \quad \mathit{proc\text{-}steps}(\mathit{seq}_{\mathrm{CVM}}, T_{\mathrm{CVM}}) = n & \\
 & \wedge\ \ \mathit{seq}\surd(\mathit{seq}_{\mathrm{CVM}}, \mathit{ds}_{\mathrm{init}}(c_{\mathrm{ISA+DS}}.\mathit{devs})) & \\
 & \wedge\ \ (\forall\, c'_{\mathrm{CVM}} : & \\
 & \qquad \delta_{\mathrm{CVM}}^{T_{\mathrm{CVM}}}(\mathit{cvm}_{\mathrm{init}}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA+DS}}.\mathit{devs}), \mathit{seq}_{\mathrm{CVM}}) = \lfloor c'_{\mathrm{CVM}} \rfloor & \\
 & \qquad \wedge\ \ \mathit{asize}_{\mathrm{heap}}(\mathit{hst}(c'_{\mathrm{CVM}}.\mathit{ak}.\mathit{mem})) \leq \mathtt{HEAP\text{-}SIZE}_{\mathrm{AK}} & \\
 & \qquad \longrightarrow\ (\exists\, T_{\mathrm{ISA}}, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}} : & \\
 & \qquad\quad \delta_{\mathrm{ISA+DS}}^{T_{\mathrm{ISA}}}(c_{\mathrm{ISA+DS}}, \mathit{seq}_{\mathrm{ISA}}) = c'_{\mathrm{ISA+DS}} & \\
 & \qquad\quad \wedge\ \ \forall\, \mathit{did} :\ c'_{\mathrm{ISA+DS}}.\mathit{devs}(\mathit{did}) \overset{\mathrm{type}}{\sim} c_{\mathrm{ISA+DS}}.\mathit{devs}(\mathit{did}) & \\
 & \qquad\quad \wedge\ \ \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
 & \qquad\quad \wedge\ \ \mathit{impl\text{-}inv}(\pi_{\mathrm{AK}}, c'_{\mathrm{CVM}}, c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}) & \\
 & \qquad\quad \wedge\ \ \mathit{cvm\text{-}sim}(\pi_{\mathrm{AK}}, c'_{\mathrm{CVM}}, c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}})\ ))). &
\end{array}
$$

Isabelle: `cvm/cvm_correct/cvm_correct.cvm_correct`

7.3 CVM Correctness Theorem Proof

The proof of the CVM correctness theorem could be split according to various types of steps that can be made in the CVM model. In this Section we present a skeleton of the proof leaving proofs of individual cases to be the topic of further chapters. In the frame of this work we consider the followng cases: context switch (Section 9.2), copy primitive(Section 9.3) and user step (Chapter 10).

We prove Theorem 7.22 by induction on N. We start every proof case by finding a right instance $\mathit{seq}_{\mathrm{CVM}}$ of a CVM sequence and T_{CVM} of a number of CVM steps. Dealing with this, we use the number of the hardware steps which are needed to complete a particular case. For the induction base we construct a CVM sequence from the ISA sequence, keeping in mind that there is no swap hard disk on the CVM level (the idle device instead), and the kernel does not make any steps. For this construction we use a number of auxiliary functions defined below. Note, that the same function will be used for proofs of individual cases of the induction step.

First of all, we introduce a function which converts ISA sequences to CVM sequences. The desired CVM sequence is obtained by replacing all system hard disk inputs by idle inputs since the hard disk is replaced by the idle device.

Definition 7.23 (Conversion of ISA sequences to CVM sequences) For a sequence *seq* used on the ISA level, the following function replaces all inputs for the system hard disk by idle inputs making the sequence suitable for usage on the CVM level:

$$\mathit{seq}_{\mathrm{ISA}} 2 \mathit{seq}_{\mathrm{CVM}}(\mathit{seq}_{\mathrm{ISA}}) \overset{\mathrm{def}}{=} \mathit{seq}_{\mathrm{CVM}},$$

where

$$seq_{\mathrm{CVM}}(t) = \begin{cases} \mathit{Dev}(\texttt{SWAP\_DID}, \mathit{idle\text{-}eifi}) & \text{if } seq_{\mathrm{ISA}}(t) = \mathit{Dev}(\texttt{SWAP\_DID}, \mathit{eifi}_{\mathrm{HD}}) \\ seq_{\mathrm{ISA}}(t) & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/cvm_correct/cvm_sequence.isa_seq_2_cvm_seq`

This conversion function guarantees us validity — liveness and well-typedness — of the resulting sequence by construction. We state this property formally in the following lemma.

Lemma 7.24 (Conversion of sequences preserves validity) A CVM sequence constructed from an ISA sequence *seq* is valid if *seq* is valid:

$$seq\surd(seq, devs) \longrightarrow seq\surd(seq_{\mathrm{ISA}}2seq_{\mathrm{CVM}}(seq), ds_{\mathrm{init}}(devs)).$$

Isabelle: `cvm/cvm_correct/cvm_sequence.live_seq_cvm_isa_seq_2_cvm_seq`

We need an operation that filters out all processor and system hard disk steps from a sequence. We define such an operation by means of three functions: the first converts a sequence into a list, the second performs the filtering, and the third converts the filtered list back to a sequence. Our motivation to this approach is that it is easier to filter a list instead of a function.

We start with a function which takes elements which are used for computations on the ISA level and packs them in a list.

Definition 7.25 (Conversion of a sequence to list) For a sequence *seq*, an initial step number T, and a number of steps N the function

$$\mathit{seq2list} :: \mathit{Seq} \times \mathbb{N} \times \mathbb{N} \mapsto \mathit{SeqEl}^*$$

constructs the list of N sequence elements starting from initial step T:

$$\mathit{seq2list}(seq, T, N) \stackrel{\mathrm{def}}{=} [seq(T), seq(T+1), \cdots, seq(T+N-1)].$$

Isabelle: `cvm/cvm_correct/cvm_sequence.seq_cut_list`

Further, we remove all elements which correspond to the swap hard disk. Surely, they are out of our interest, since on the CVM level the swap hard disk is replaced by the idle device. Steps of the idle device do not influence the whole computation. The function defined below also removes processor steps from the sequence because such ISA computation corresponds at most to the one kernel step on the CVM level.

Definition 7.26 (Remove processor and system hard disk steps) For a list of sequence elements seq_{list} the function

$$\mathit{dev\text{-}wo\text{-}hd} :: \mathit{SeqEl}^* \mapsto \mathit{SeqEl}^*$$

removes all elements corresponding to the processor or the system hard disk:

$$\mathit{dev\text{-}wo\text{-}hd}(seq_{\mathrm{list}}) \stackrel{\mathrm{def}}{=} [x_{seq_{\mathrm{list}}} : x \neq \mathit{Proc} \wedge x \neq \mathit{Dev}(\texttt{SWAP\_DID}, \mathit{idle\text{-}eifi})].$$

Isabelle: `cvm/cvm_correct/cvm_sequence.dev_list_wo_hd`

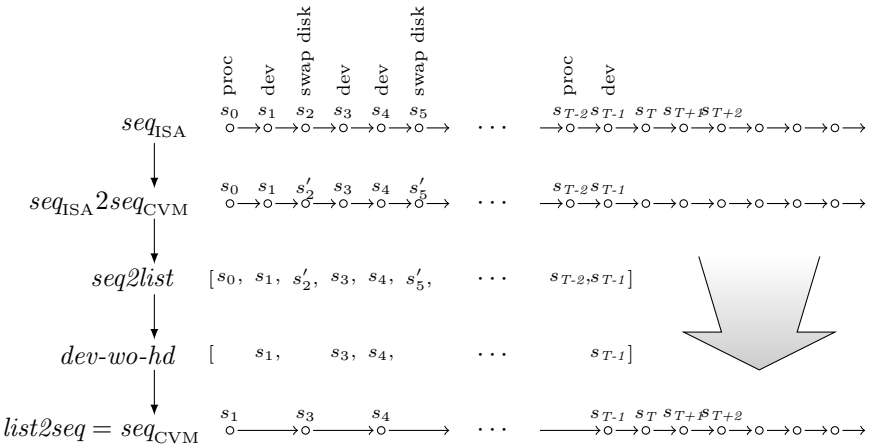

Figure 7.4: Creating the CVM sequence from ISA sequence. Initial case.

Finally, we define a function which converts a list of elements to a sequence by updating a prefix of a sequence with a given list of elements.

Definition 7.27 (Sequence prefix update) For a sequence seq and a list of sequence elements seq_{list} the function

$$list2seq :: Seq \times \mathbb{N} \times SeqEl^* \mapsto Seq$$

updates the first $|seq_{\text{list}}|$ elements of the sequence by elements from the list starting from the position T:

$$list2seq(seq, T, seq_{\text{list}}) \stackrel{\text{def}}{=} seq',$$

where

$$seq'(t) = \begin{cases} seq_{\text{list}}[t-T] & \text{if } T \leq t < T + |seq_{\text{list}}| \\ seq(t) & \text{otherwise} \end{cases}.$$

Isabelle: `cvm/cvm_correct/cvm_sequence.live_cvm_seq`

The following lemma justifies validity preservation under the three functions defined above.

Lemma 7.28 (Filtering preserves validity) The validity of the sequence updated by the list with removed processor and hard disk elements follows from the validity of the initial sequence:

$$seq_{\surd}(seq, devs) \longrightarrow seq_{\surd}(list2seq(seq, T, dev\text{-}wo\text{-}hd(seq2list(seq, T, N))), devs).$$

Isabelle: `cvm/cvm_correct/cvm_sequence.live_seq_cvm_live_cvm_seq`

Figure 7.4 reflects the construction process for the initial case. For the induction base we instantiate the CVM sequence as follows:

$$seq_{\text{CVM}} = list2seq(seq_{\text{ISA}}2seq_{\text{CVM}}(seq_{\text{ISA}}), 0, dev\text{-}wo\text{-}hd(seq2list(seq_{\text{ISA}}, 0, T_{\text{ISA}}))).$$

The CVM steps T_{CVM} are equal to the length of the created list:

$$T_{\mathrm{CVM}} \;=\; |\mathit{dev\text{-}wo\text{-}hd}(\mathit{seq2list}(\mathit{seq}_{\mathrm{ISA}}, 0, T_{\mathrm{ISA}}))|.$$

It is not difficult to show that the number of processor steps equals to zero since we have removed all such elements. Note that construction of the CVM sequence depends on the number of ISA steps which we get for every case from the low level correctness proof.

While dealing with the induction step, we already have a CVM sequence $\mathit{seq}_{\mathrm{CVM}}$ from the induction hypothesis. Moreover, this sequence is live and well-typed. The sequence satisfies necessary criteria up to the N-th kernel step. For the $(N+1)$-th step we, though, need to update a corresponding part of the sequence. We also have a number T_{CVM} of CVM steps which contains N kernel steps, and a number T_{ISA} of corresponding hardware steps. For the current, $(N+1)$-th, step let there be additional T'_{ISA} steps of the underlying ISA with devices. Our goal is to consider the last part of the hardware sequence which corresponds to T'_{ISA} steps and to construct the necessary part $\mathit{seq}'_{\mathrm{CVM}}$ of the CVM sequence out of it. The devices steps in the hardware sequence are projected to the CVM sequence by filtering out steps of the system hard disk. As for the processor steps, two cases are possible.

For all other but user step cases, e.g., non-device primitive execution, or switch between the kernel and some process all processor steps correspond to a single kernel step. It does not matter at which point we insert this processor step in the obtained sequence of devices step. Our choice is to add the processor step at the end of the list:

$$\mathit{seq}'_{\mathrm{CVM}} \;=\; \mathit{list2seq}(\mathit{seq}_{\mathrm{CVM}}, T_{\mathrm{CVM}}, \mathit{dev\text{-}wo\text{-}hd}(\mathit{seq2list}(\mathit{seq}_{\mathrm{ISA}}, T_{\mathrm{ISA}}, T'_{\mathrm{ISA}})) \circ [\mathsf{Proc}]).$$

The situation is different for the user steps. We cannot arbitrarily reorder the devices since their states might influence user executions: interrupts from devices must be considered at the point when a user process makes a step. Because of that we have divided the proof of the user step into two lemmas: (i) handling of page faults that might occur during the user run and execution of devices such that the next step will be done by the user, and (ii) the possible kernel execution together with devices in case of interrupts. So, we have two numbers of hardware steps corresponding to these cases: T^1_{ISA} and T^2_{ISA}. Note, that T^1_{ISA} could, possibly, be zero, and T^2_{ISA} starts from a processor step. Then the sequence is updated as follows:

$$\begin{aligned}\mathit{seq}'_{\mathrm{CVM}} \;=\; \mathit{list2seq}(&\mathit{seq}_{\mathrm{CVM}}, T_{\mathrm{CVM}},\\ &\mathit{dev\text{-}wo\text{-}hd}(\mathit{seq2list}(\mathit{seq}_{\mathrm{ISA}}, T_{\mathrm{ISA}}, T^1_{\mathrm{ISA}}))\\ &\quad\circ [\mathsf{Proc}]\\ &\quad\circ \mathit{dev\text{-}wo\text{-}hd}(\mathit{seq2list}(\mathit{seq}_{\mathrm{ISA}}, T_{\mathrm{ISA}} + T^1_{\mathrm{ISA}}, T^2_{\mathrm{ISA}}))).\end{aligned}$$

In both cases we can easily show that the number of the kernel steps is equal to one. So if we have some previous CVM computation

$$\delta^{T_{\mathrm{CVM}}}_{\mathrm{CVM}}(\mathit{cvm}_{\mathrm{init}}(\pi_{\mathrm{AK}}, \mathit{devs}), \mathit{seq}_{\mathrm{CVM}}) = \lfloor c_{\mathrm{CVM}} \rfloor,$$

then the computation in the induction step

$$\delta^{T_{\mathrm{CVM}}+T'_{\mathrm{CVM}}}_{\mathrm{CVM}}(\mathit{cvm}_{\mathrm{init}}(\pi_{\mathrm{AK}}, \mathit{devs}), \mathit{seq}'_{\mathrm{CVM}}) = \lfloor c'_{\mathrm{CVM}} \rfloor$$

differs in all elements except for the devices only in one kernel step, as

$$\delta^{1}_{\mathrm{CVM}}(c_{\mathrm{CVM}}, \mathit{seq}) = \lfloor c'_{\mathrm{CVM}} \rfloor$$

with $seq(0) = \mathit{Proc}$.

Most of the theorem's conclusions follow directly from assumptions by simple rewriting. The kernel relation $\mathit{kernel\text{-}rel}(c'_{\mathrm{CVM}}.ak, c'_{\mathrm{C0}}, \pi_{\mathrm{AK}})$ which is a part of CVM simulation relation $\mathit{cvm\text{-}sim}(\pi_{\mathrm{AK}}, c'_{\mathrm{CVM}}, c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}})$ is proven basically from two facts. First, in all cases except the abstract kernel step, the global and the heap memory might be changed only in the part which belongs to the CVM program. Hence, the value of the global and heap variables stay the same. Second, even the execution of the primitives do not destroy the local stack, and the set of initialized variables as well as the heap table could only grow. We proceed with formal definitions of these properties. First, we introduce a predicate that compares two C0 memory configurations and asserts that they differ only at given global and heap variable names (locations).

Definition 7.29 (Unchanged global and heap memories) Let mc and mc' be C0 memory configurations, let $\mathit{vars}_{\mathrm{gm}}$ be a set of names of global variables that might be changed during a transition from mc to mc', and let $\mathit{ind\text{-}set}_{\mathrm{heap}}$ be a set of indices of heap variables that might be changed. The following predicate claims that evaluation of all variables except for those that are in the sets $\mathit{vars}_{\mathrm{gm}}$ and $\mathit{ind\text{-}set}_{\mathrm{heap}}$ has equal results in memory configurations mc and mc':

$$
\begin{aligned}
&\mathit{unchanged}_{\mathrm{GM,HM}}(te, mc, mc', \mathit{vars}_{\mathrm{gm}}, \mathit{ind\text{-}set}_{\mathrm{heap}}) \stackrel{\mathrm{def}}{=} \\
&\qquad \forall\ vn \in (\mathit{vns}(\mathit{gst}(mc)) \cap mc.gm.init) \setminus \mathit{vars}_{\mathrm{gm}} : \\
&\qquad\qquad \mathit{reval}(te, mc', \mathit{gvar}_{\mathrm{gm}}(vn)) = \mathit{reval}(te, mc, \mathit{gvar}_{\mathrm{gm}}(vn)) \\
&\qquad \wedge \quad \forall\ i \in \{j :\ j < |\mathit{hst}(mc)| \wedge j \notin \mathit{ind\text{-}set}_{\mathrm{heap}}\} : \\
&\qquad\qquad \mathit{reval}(te, mc', \mathit{gvar}_{\mathrm{hm}}(i)) = \mathit{reval}(te, mc, \mathit{gvar}_{\mathrm{hm}}(i)).
\end{aligned}
$$

Isabelle: `C0SS/MoreC0.gm_hm_unchanged`

With the help of the above predicate we can state that the concrete kernel changes only global and heap variables from the CVM implementation part and does not touch the abstract kernel variables.

Definition 7.30 (Unchanged abstract global and heap variables) Let c_{C0} and c'_{C0} be two C0 configurations. The predicate $\mathit{abs\text{-}kern\text{-}unch}_{\mathrm{GM,HM}}$ holds if only the CVM global variables obtained as $\mathit{vns}(\mathit{gst}_{\mathrm{CVM}})$ and the heap variables occupied by heap indices up to `CVM_HST_LEN` are changed. Formally, the predicate is as follows:

$$
\begin{aligned}
&\mathit{abs\text{-}kern\text{-}unch}_{\mathrm{GM,HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) \stackrel{\mathrm{def}}{=} \\
&\qquad \mathit{unchanged}_{\mathrm{GM,HM}}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\mathit{mem}, c'_{\mathrm{C0}}.\mathit{mem}, \\
&\qquad\qquad \mathit{vns}(\mathit{gst}_{\mathrm{CVM}}), \{i :\ i < \texttt{CVM\_HST\_LEN}\}).
\end{aligned}
$$

Isabelle: `cvm/config/kernel_separate.abstract_gm_hm_unchanged`

Next, we define a predicate which is intended to specify C0 memory configurations after execution of a call statement.

Definition 7.31 (C0 memory invariant) Let mc and mc' be memory configurations before and after execution of some call statement, let st_{heap} be an additional part of the heap symbol table which has been allocated during this call, and let *ret-var* and *ret-val* be a name and a value of the return variable of the call, respectively. The predicate below

which we call a *C0 memory invariant* is a conjunction of the following facts: (i) global symbol tables of both memory configurations are equal while the set of initialized global variables of mc is included in that of mc', (ii) the lengths of local memory stacks are equal and these stack differ only at the topmost element, (iii) top return destinations and top local symbol tables of memory configurations mc and mc' are equal while the set of initialized top local variables of mc is a part of same kind of set of mc', (iv) all initialized top local variables have the same values in both memory configurations except the return variable, (v) the value of the return variable *ret-var* is *ret-val*, and (vi) heap symbol table of mc' is combined out of the heap symbol table of mc and st_{heap}. Formally:

$$
\begin{aligned}
&\textit{is-C0mem-inv}(te, mc, mc', st_{\text{heap}}, \textit{ret-var}, \textit{ret-val}) \stackrel{\text{def}}{=} \\
&\qquad \textit{gst}(mc') = \textit{gst}(mc) \\
&\quad \wedge \quad mc.gm.init \subseteq mc'.gm.init \\
&\quad \wedge \quad |mc'.lm| = |mc.lm| \\
&\quad \wedge \quad \forall\, i < |mc.lm| - 1 :\ mc'.lm[\![i]\!] = mc.lm[\![i]\!] \\
&\quad \wedge \quad \textit{res}_{\text{top}}(mc') = \textit{res}_{\text{top}}(mc) \\
&\quad \wedge \quad \textit{lst}_{\text{top}}(mc') = \textit{lst}_{\text{top}}(mc) \\
&\quad \wedge \quad \textit{lm}_{\text{top}}(mc).\textit{init} \cup \{\textit{ret-var}\} \subseteq \textit{lm}_{\text{top}}(mc').\textit{init} \\
&\quad \wedge \quad \forall\, vn \in (\textit{vns}(\textit{lst}_{\text{top}}(mc)) \cap \textit{lm}_{\text{top}}(mc).\textit{init}) \setminus \{\textit{ret-var}\} : \\
&\qquad\qquad \textit{reval}(te, mc', \textsf{gvar}_{\text{lm}}(|mc.lm| - 1, vn)) = \\
&\qquad\qquad \textit{reval}(te, mc, \textsf{gvar}_{\text{lm}}(|mc.lm| - 1, vn)) \\
&\quad \wedge \quad \textit{reval}(te, mc', \textsf{gvar}_{\text{lm}}(|mc.lm| - 1, \textit{ret-var})) = \textit{ret-val} \\
&\quad \wedge \quad \textit{hst}(mc') = \textit{hst}(mc) \circ st_{\text{heap}}.
\end{aligned}
$$

Isabelle: `C0SS/MoreC0.structure_preserved`

There is one more definition which we will use during the verification of function call statements. After an execution of a call statement the program rest is changed in a way that the call statement is replaced by the empty statement. For this operation we introduce the following function.

Definition 7.32 (C0 program rest invariant) The function

$$\textit{rem-1st-stmt} :: \textit{Stmt} \mapsto \textit{Stmt}$$

is defined inductively over the statements and modifies them in the following way. For the call statement it returns *skip*:

$$\textit{rem-1st-stmt}(\textsf{sCall}(vn, fn, param, sid)) = \textsf{skip}.$$

For the compositional statement it goes deeper through the statement tree:

$$\textit{rem-1st-stmt}(\textsf{comp}(s_1, s_2)) = \textsf{comp}(\textit{rem-1st-stmt}(s_1), s_2).$$

The function has no effect on all remaining statements.

Isabelle: `C0SS/MoreC0.remove_first_non_Skip`

Chapter 8

Formal Inline-Assembly Semantics

Contents

In this chapter we introduce an approach to formal reasoning about inline-assembly portions in C0 programs. There are several possibilities to argue about correctness of programs with parts written in assembly. One might have an idea to compile such programs and formulate correctness theorems about their object code in the machine-language semantics. As long as such theorems are proven interactively the approach falls short when dealing with non trivial programs, like our kernel, with the object code of some thousands lines. Another possibility, actually chosen by a number of OS verification projects [113, 46, 36, 56], is to rely on unsafe portions of assembly code. This badly contradicts the principles of pervasive verification. In contrast, our approach provides effective means to reason about C0 programs with inline-assembly parts not at the cost of pervasiveness.

The chapter starts in Section 8.1 with an overview of our approach to verification of inline-assembly statements. In Section 8.2 we list the necessary restrictions over inline-assembly portions that make their formal verification possible in our context. Section 8.3 introduces a function which projects the effects of assembly portions to the C0 level and updates respective C0 variables. Further, in Section 8.4 we analyze the ranges occupied in C0 and assembly memory by the variables updated during executions of inline assembly parts. In Section 8.5 we prove correctness of our approach.

8.1 Overview

Section 4.2 introduces $\mathit{consis}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, \mathit{alloc}, c_{\mathrm{ASM}})$, a C0 compiler simulation relation towards VAMP assembly. The C0 compiler correctness theorem (Theorem 4.10) proves

Table 8.1: Test predicates for g-variables.

g	*is-gm-gvar*(g)	*is-lm-gvar*$_{\text{top}}(mc, g)$	*is-hm-gvar*(g)
$\mathit{gvar}_{\text{gm}}(vn)$	T	F	F
$\mathit{gvar}_{\text{lm}}(i, vn)$	F	$i + 1 = \lvert mc.lm \rvert$	F
$\mathit{gvar}_{\text{hm}}(i)$	F	F	T
$\mathit{gvar}_{\text{arr}}(g', i)$	*is-gm-gvar*(g')	*is-lm-gvar*$_{\text{top}}(mc, g')$	*is-hm-gvar*(g')
$\mathit{gvar}_{\text{str}}(g', cn)$	*is-gm-gvar*(g')	*is-lm-gvar*$_{\text{top}}(mc, g')$	*is-hm-gvar*(g')

this relation for all but inline-assembly statements. When verifying a C0_A program as long as no assembly statement $\mathit{asm}(il)$ occurs the C0 semantics is applied. The original approach to deal with verification of an assembly statement was to maintain the compiler consistency relation with execution of every single instruction from *il* [41]. This method turned out to be inconvenient due to excessive complexity of formal proofs, therefore a new one was developed and used [106].

Briefly, our approach to deal with inline-assembly statements is as follows. On an assembly statement the execution is switched to the consistent assembly configuration and continues directly there. When the assembly instructions have been executed we switch back to the C0 level. For this we have to update the C0 configuration possibly affected by the assembly instructions. An allocation function *alloc* makes it possible to determine which variables of the C0 configuration have changed. We retrieve their values from the assembly and write back to the C0 memory configuration.

8.2 Preconditions

A number of restrictions are imposed on the inline-assembly computation, which guarantees that the C0 configuration is not destroyed by the assembly portion.

Before we list these restrictions let us recall several definitions from the C0 small-step semantics. The code of the program specified by a type environment *te*, function table *ft*, and global-symbol table *gst* is generated by means of the function $\mathit{code}_{\text{prog}}(te, ft, gst)$[66, Definition 7.36]. We denote that a g-variable g of a memory configuration mc is reachable by $g \in \mathit{reachable}_{\text{g}}(mc)$[66, Definition 8.3]. That g is a root g-variable is denoted by $\mathit{root}_{\text{g}}(g)$[66, Definition 4.11]. A g-variable g is called initialized, denoted by $\mathit{initialized}_{\text{g}}(mc, g)$, if its root g-variable is in the set of initialized variables of the corresponding memory frame mc [66, Definition 4.20]. The base address of the global memory is computed by the function $\mathit{abase}_{\text{gm}}(te, ft, gst)$[66, Definition 7.15]. The allocated size of the heap for a symbol table st is computed by the function $\mathit{asize}_{\text{heap}}(st)$[66, Definition 7.12]. Three test predicates defined in Table 8.1 — *is-gm-gvar*(g), *is-lm-gvar*$_{\text{top}}(mc, g)$, and *is-hm-gvar*(g) — are used to check whether g is a global, top local, or heap g-variable.

Definition 8.1 (Preconditions to inline-assembly statement) Let *te* be a type environment, let *ft* be a function table, let c_{C0} be a C0 configuration with assembly statement to be executed next $\mathit{stmt}(c_{\text{C0}}) = \mathit{asm}(il, sid)$. Let c_{ASM} and c'_{ASM} be assembly configurations before and after the execution of the instructions *il*, let *alloc* be a C0 allocation function. Let *gl* be a list of g-variables to be updated during the execution of *il*. The predicate *precond-C0-asm-upd*$(te, ft, c_{\text{C0}}, c_{\text{ASM}}, c'_{\text{ASM}}, \mathit{alloc}, gl)$ collects necessary preconditions for the construction of an updated C0 configuration after an execution of

the inline-assembly code *il*:

- general-purpose registers used by the C0 compiler, namely the stack pointer r_{sbase}, heap pointer r_{htop}, and top-local pointer r_{lframe}, stay unchanged:

$$\forall\, r \in \{r_{\text{sbase}}, r_{\text{htop}}, r_{\text{lframe}}\} :\ c'_{\text{ASM}}.gpr[r] = c_{\text{ASM}}.gpr[r],$$

- program counters point to the end of the assembly portion *il*:

$$c'_{\text{ASM}}.dpc = c_{\text{ASM}}.dpc + 4 \cdot |il| \ \wedge\ c'_{\text{ASM}}.pc = c'_{\text{ASM}}.dpc + 4,$$

- the memory region where the code lies stay unchanged, i.e., we forbid self-modifying code — let us abbreviate the code length as $len = csize_{\text{prog}}(te, gst(c_{\text{C0}}.mem), ft)$:

$$\textit{get-data}(c_{\text{ASM}}.m', \texttt{PROGBASE}, len) = \textit{get-data}(c_{\text{ASM}}.m, \texttt{PROGBASE}, len),$$

- all variable of *gl* are reachable in the memory configuration $c_{\text{C0}}.mem$:

$$\forall\, g \in gl :\ g \in \textit{reachable}_{\text{g}}(c_{\text{C0}}.mem),$$

- all variable of *gl* are of some elementary type — complex variables could be updated through their elementary components:

$$\forall\, g \in gl :\ \textit{is-elem}_{\text{t}}(ty_{\text{g}}(te, sc(c_{\text{C0}}.mem), g)),$$

- only global, top local, and heap variables are allowed to be updated:

$$\forall\, g \in gl :\ \textit{is-gm-gvar}(g)\ \vee\ \textit{is-lm-gvar}_{\text{top}}(c_{\text{C0}}.mem, g)\ \vee\ \textit{is-hm-gvar}(g),$$

- all variables of *gl* are either root or initialized g-variables — C0 semantics does not allow to initialize subvariables of a not initialized variable

$$\forall\, g \in gl :\ \textit{initialized}_{\text{g}}(c_{\text{C0}}.mem, g)\ \vee\ \textit{root}_{\text{g}}(g),$$

- only those variables that are given by the list *gl* are changed:

$$\begin{array}{lll} \forall\, a : & & \textit{abase}_{\text{gm}}(te, ft, gst(c_{\text{C0}}.mem)) < a \\ & \wedge & a < \texttt{ABASE}_{\text{hm}} + \textit{asize}_{\text{heap}}(hst(c_{\text{C0}}.mem)) \\ & \wedge & \forall\, g \in gl :\ a/4 \neq \textit{alloc}(g).b/4 \\ & \longrightarrow & c'_{\text{ASM}}.m_{\textit{word}}(a) = c_{\text{ASM}}.m_{\textit{word}}(a). \end{array}$$

Isabelle: `C0ASS/asm_stmt_step`

8.3 Updating C0 Configurations

Definition 8.2 (Memory cell construction) For a given integer number z and a type *ty* the function $\textit{mcell-cons}(ty, z)$ constructs a C0 memory cell of type *ty*. It yields some memory cell $\lfloor mcell \rfloor$ only if *ty* is an elementary type, i.e., $ty \in \{\textit{bool}_{\text{T}}, \textit{char}_{\text{T}}, \textit{unsgnd}_{\text{T}},$

$\mathsf{ptr}_{\mathrm{T}}(pn)\}$, and the value $\bot$, otherwise. The memory cell *mcell* is obtained by type casting z to the corresponding value of the type *ty*.

$$\mathit{mcell\text{-}cons}(ty, z) \stackrel{\text{def}}{=} \begin{cases} \lfloor \mathsf{bool}(\mathsf{F}) \rfloor & \text{if } ty = \mathsf{bool}_{\mathrm{T}} \wedge z = 0 \\ \lfloor \mathsf{bool}(\mathsf{T}) \rfloor & \text{if } ty = \mathsf{bool}_{\mathrm{T}} \wedge z = 1 \\ \lfloor \mathsf{int}(z) \rfloor & \text{if } ty = \mathsf{int}_{\mathrm{T}} \\ \lfloor \mathsf{nat}(i2n(z)) \rfloor & \text{if } ty = \mathsf{unsgnd}_{\mathrm{T}} \\ \lfloor \mathsf{char}(z) \rfloor & \text{if } ty = \mathsf{char}_{\mathrm{T}} \wedge -2^7 \leq z < 2^7 \\ \lfloor \mathsf{ptr}(\bot) \rfloor & \text{if } ty = \mathsf{ptr}_{\mathrm{T}}(tn) \wedge z = 0 \\ \bot & \text{otherwise} \end{cases}$$

Isabelle: `C0ASS/asm_stmt_step.elem_memcell_construction`

Recall, that the type of a g-variable g with respect to a symbol configuration sc is computed by means of the function $ty_{\mathrm{g}}(sc, g)$[66, Definition 4.15]. A memory configuration mc is updated at a g-variable g with a value v by means of the function $\mathit{memupd}(mc, g, v)$[66, Section 4.4.2].

Definition 8.3 (G-variable update) Let te be a type environment, let mc be a C0 memory configuration, let c'_{ASM} be an assembly configuration, let *alloc* be an allocation function, and let g be a g-variable. The function $\mathit{C0\text{-}asm\text{-}upd}_{\mathrm{gvar}}(te, mc, c'_{\mathrm{ASM}}, \mathit{alloc}, g)$ updates the memory configuration mc at the variable g as follows. Let $z = c'_{\mathrm{ASM}}.m_{\mathit{word}}(\mathit{alloc}(g).b)$ be an integer value retrieved from the memory of the assembly configuration c'_{ASM} at the allocated address of g. We construct a memory cell of the type of g storing the value z by $\mathit{mcell\text{-}cons}(ty_{\mathrm{g}}(mc, g), z)$. If the construction ends up in an error state $\bot$, so the g-variable update function does. Otherwise the result of the construction is some memory cell $\lfloor \mathit{mcell} \rfloor$. Then g-variable update function is defined as:

$$\mathit{C0\text{-}asm\text{-}upd}_{\mathrm{gvar}}(te, mc, c'_{\mathrm{ASM}}, \mathit{alloc}, g) \stackrel{\text{def}}{=} \lfloor \mathit{memupd}(mc, g, \mathit{mcell}) \rfloor.$$

Isabelle: `C0ASS/asm_stmt_step.C0_mem_asm_update_gvar`

The C0-memory update function $\mathit{C0\text{-}asm\text{-}upd}_{\mathrm{mem}}(te, mc, c'_{\mathrm{ASM}}, \mathit{alloc}, gl)$ is defined by induction over the list g-variables gl. For the base case $gl = []$ the function returns $\lfloor mc \rfloor$. In the induction step $gl = g \circ gl'$ the variable g is updated by means $\mathit{C0\text{-}asm\text{-}upd}_{\mathrm{gvar}}(te, mc, c'_{\mathrm{ASM}}, \mathit{alloc}, g)$. If this update results in an error $\bot$, so the induction step does. Otherwise, the g-variable update yields some new C0 memory configuration mc' and we define the induction step as $\mathit{C0\text{-}asm\text{-}upd}_{\mathrm{mem}}(te, mc', c'_{\mathrm{ASM}}, \mathit{alloc}, gl')$.

Definition 8.4 (C0-configuration update) Let te be a type environment, let *ft* be a function table, let c_{C0} be a C0 small-step semantics configuration, let c_{ASM} and c'_{ASM} be assembly configurations, let *alloc* be an allocation function, and let gl be a list of g-variables. In case the preconditions to inline assembly statement $\mathit{precond\text{-}C0\text{-}asm\text{-}upd}$ $(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}}, c'_{\mathrm{ASM}}, \mathit{alloc}, gl)$ are not satisfied or C0-memory update function $\mathit{C0\text{-}asm\text{-}upd}_{\mathrm{mem}}(te, c_{\mathrm{C0}}.\mathit{mem}, c'_{\mathrm{ASM}}, \mathit{alloc}, gl)$ ends up in an error state $\bot$, so the C0-configuration update function does. Otherwise the result of the memory update yields some new memory configuration $\lfloor mc \rfloor$ which is used to update the given c0 configuration:

$$\mathit{C0\text{-}asm\text{-}upd}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}}, c'_{\mathrm{ASM}}, \mathit{alloc}, gl) \stackrel{\text{def}}{=} \lfloor c'_{\mathrm{C0}} \rfloor,$$

Table 8.2: Base address of a g-variable g and size of its type ty.

	base address	size of type	definitions in [66]
in C0	$ba_{\mathrm{g}}(sc, g)$	$size_{\mathrm{t}}(ty)$	Definitions 4.18, 4.1
allocated in assembly	$abase_{\mathrm{g}}(te, ft, sc, g)$	$asize_{\mathrm{t}}(ty)$	Definitions 7.17, 7.10

where the memory component of the updated configuration c'_{C0} is defined as $c'_{\mathrm{C0}}.mem = mc$ and the first statement of the program rest $c'_{\mathrm{C0}}.prog$ is replaced with *skip*.

Isabelle: `C0ASS/asm_stmt_step.C0_conf_asm_update`

8.4 Range Analysis for Elementary Variables

Having a function that projects effects of assembly portions to the C0 level we aim at proving its correctness. The correctness notion is given by the simulation relation $consis(te, ft, c_{\mathrm{C0}}, alloc, c_{\mathrm{ASM}})$ between C0 and assembly. The data-consistency term of this relation requires, among others, that all elementary variables (i.e., variables of elementary types) occupy different C0 memory cells and different assembly memory cells. In order to prove this we investigate the ranges that elementary variables occupy in memories of C0 and assembly configurations.

Let g be a g-variable of type $ty = ty_{\mathrm{g}}(sc, g)$. Table 8.2 collects functions which are used to obtain the base addresses of g and sizes of type ty. There are two versions of each definition: the first computes these notions in the C0 memory configuration while the second computes their versions allocated in the assembly memory. The former is measured in cells, the latter in bytes. Using these definitions we introduce the notion of a variable range, a space which a g-variable occupies in memory. Formally, a variable range is a pair constituting the variable's base address and the size of its type. The functions $range_{\mathrm{g}}^{\mathrm{ASM}}$ and $range_{\mathrm{g}}^{\mathrm{C0}}$ define a variable range in the C0 and assembly memory respectively:

$$\begin{aligned} range_{\mathrm{g}}^{\mathrm{ASM}}(te, ft, sc, g) &\stackrel{\mathrm{def}}{=} (ba_{\mathrm{g}}(sc, g), size_{\mathrm{t}}(ty_{\mathrm{g}}(sc, g))), \\ range_{\mathrm{g}}^{\mathrm{C0}}(te, ft, sc, g) &\stackrel{\mathrm{def}}{=} (abase_{\mathrm{g}}(te, ft, sc, g), asize_{\mathrm{t}}(ty_{\mathrm{g}}(sc, g))). \end{aligned}$$

However, the analysis presented in this section is common to both C0 and assembly variable ranges. Therefore, we will simply use the notation $range_{\mathrm{g}}(te, ft, sc, g)$ — the reader can substitute the appropriate definition for C0 or assembly instead.

For a g-variable g let $level(g)$ be its level, a natural number measuring the recursion depth of its constructors. The set of a g-variable's sub variables is defined as $sub_{\mathrm{g}}(g)$[66, Definition 4.12].

Lemma 8.5 (Parent exists) Let a be a g-variable, let i and n be natural numbers. If n is the level of a and i is less than n, then there exists a variable b with the level i such that a is a sub-variables of b:

$$level(a) = n \ \wedge \ i < n \ \longrightarrow \ \exists\, b : \ level(b) = i \ \wedge \ a \in sub_{\mathrm{g}}(b).$$

Isabelle: `C0Acompilersimulation/abase_lemmata.parent_exists`

Proof. By induction on n.

Base: $n = 0$. The assumptions are falsified due to $i < 0$.

Step: $n \to n+1$. We have to prove: $\mathit{level}(a) = n+1 \ \wedge \ i < n+1 \ \longrightarrow \ \exists \ b : \mathit{level}(b) = i \ \wedge \ a \in \mathit{sub}_{\mathrm{g}}(b)$. Since $\mathit{level}(a) > 0$ variable a could only be an array element $a = \mathit{gvar}_{\mathrm{arr}}(g, j)$ or a structure component $a = \mathit{gvar}_{\mathrm{str}}(g, cn)$. Obviously, $\mathit{level}(g) = n$ and $a \in \mathit{sub}_{\mathrm{g}}(g)$.

Case $i = n$. Instantiating the existentially quantified b with g we conclude $\mathit{level}(g) = n = i$ and $a \in \mathit{sub}_{\mathrm{g}}(g)$.

Case $i \neq n$. We use the induction hypothesis for g. The assumptions of the hypothesis hold, the existential quantifier is eliminated by introducing a new free variable b', hence we have $\mathit{level}(b') = i \wedge g \in \mathit{sub}_{\mathrm{g}}(b')$. We instantiate $\exists$ in the conclusion with b'. From $a \in \mathit{sub}_{\mathrm{g}}(g)$ and $g \in \mathit{sub}_{\mathrm{g}}(b')$ we conclude $a \in \mathit{sub}_{\mathrm{g}}(b')$.

□

For pairs of natural numbers a and b representing ranges we denote by $b \subseteq a$ that b is completely contained in a, by $a \asymp b$ that a and b are disjoint, and by $a \not\asymp b$ that a and b overlap.

Lemma 8.6 (Transitivity of $\subseteq$) For ranges p, q and r the transitivity property holds:

$$p \subseteq q \wedge q \subseteq r \longrightarrow p \subseteq r.$$

Isabelle: `C0Acompilersimulation/abase_lemmata.range_contains_trans`

In the remainder of this section we will show properties of pairs of global variables.

For the following lemmas (Lemma 8.7–8.10) let te be a type environment, let ft be a function table, let mc be a C0 memory configuration, and let a and b be g-variables. All mentioned lemmas use the set of common assumptions which we present only once below and do not write in the statements of lemmas explicitly. Without loss of generality let us assume that both g-variables are global:

$$\mathit{is\text{-}gm\text{-}gvar}(a) \ \wedge \ \mathit{is\text{-}gm\text{-}gvar}(b).$$

Further, let their validity hold:

$$a \in \mathit{gvars}\surd(\mathit{sc}(mc)) \ \wedge \ b \in \mathit{gvars}\surd(\mathit{sc}(mc))$$

as well as the validity of the global-symbol table of mc:

$$\mathit{gst}(mc) \in \mathit{valid}_{\mathrm{st}}(te).$$

Lemma 8.7 (Range contains ranges of sub variables) Assume that a is a sub variable of b, then the range of a is contained inside the range of b:

$$a \in \mathit{sub}_{\mathrm{g}}(b) \ \longrightarrow \ \mathit{range}_{\mathrm{g}}(te, ft, mc, a) \subseteq \mathit{range}_{\mathrm{g}}(te, ft, mc, b).$$

Isabelle: `C0Acompilersimulation/abase_lemmata.sub_gvars_impl_range_contains_abase_gl` (asm)
`C0Acompilersimulation/abase_lemmata.sub_gvars_impl_range_contains` (C0)

Proof. By induction on definition of $sub_{\mathrm{g}}(b)$.

Base: $a = b$. By the identity property of $\subseteq$.

Step: $h \in sub_{\mathrm{g}}(b) \ \wedge \ (a = \mathit{gvar}_{\mathrm{arr}}(h, i) \ \vee \ a = \mathit{gvar}_{\mathrm{str}}(h, cn))$. From the induction hypothesis we have $range_{\mathrm{g}}(te, ft, mc, h) \subseteq range_{\mathrm{g}}(te, ft, mc, b)$. Since all components of a complex variable have their range inside the range of this variable we have $range_{\mathrm{g}}(te, ft, mc, a) \subseteq range_{\mathrm{g}}(te, ft, mc, h)$. We conclude the goal by transitivity (Lemma 8.6).

□

Lemma 8.8 (Variables at the same level have disjoint ranges) Assume that a and b are different g-variable with the same level, then their ranges are disjoint:

$$a \neq b \ \wedge \ level(a) = level(b) \ \longrightarrow \ range_{\mathrm{g}}(te, ft, mc, a) \asymp range_{\mathrm{g}}(te, ft, mc, b).$$

Isabelle: `C0Acompilersimulation/abase_lemmata.same_level_impl_not_range_overlap_abase_gl` (asm)
`C0Acompilersimulation/abase_lemmata.same_level_impl_not_range_overlap` (C0)

Proof. By induction on $n = level(a) = level(b)$.

Base: $n = 0$. Both a and b are root g-variables and defined in the global-symbol table. It is easy to show by induction on the symbol table that address of the next variable is greater than address of the previous one plus the size of its type.

Step: $n \rightarrow n + 1$. The variables could be an array element $a = \mathit{gvar}_{\mathrm{arr}}(a', i_a)$, $b = \mathit{gvar}_{\mathrm{arr}}(b', i_b)$ or a structure component $a = \mathit{gvar}_{\mathrm{str}}(a', cn_a)$, $b = \mathit{gvar}_{\mathrm{str}}(b', cn_b)$. Clearly, $level(a') = level(b') = n$.

Case $a' = b'$. Both a and b belong to the same complex structure, and the goal follows by correctness of the algorithm which computes an offset inside the complex variable.

Case $a' \neq b'$. The variables belong to different complex variables. From the hypothesis we have $range_{\mathrm{g}}(te, ft, mc, a') \asymp range_{\mathrm{g}}(te, ft, mc, b')$. By Lemma 8.7 we obtain $range_{\mathrm{g}}(te, ft, mc, a) \subseteq range_{\mathrm{g}}(te, ft, mc, a')$ as well as the same condition for b' and b. From that we conclude the goal.

□

Lemma 8.9 (Elementary variables are disjoint from variables at higher level) Assume that a is of an elementary type and that a has a level below the level of b, then rages of a and b are disjoint:

$$\begin{aligned} &\mathit{is\text{-}elem}_{\mathrm{t}}(ty_{\mathrm{g}}(te, sc(mc), a)) \ \wedge \ level(a) < level(b) \\ &\longrightarrow \ range_{\mathrm{g}}(te, ft, mc, a) \asymp range_{\mathrm{g}}(te, ft, mc, b) \end{aligned}$$

Isabelle: `C0Acompilersimulation/abase_lemmata.not_range_overlap_with_elementary_diff_level_abase_gl` (asm)
`C0Acompilersimulation/abase_lemmata.not_range_overlap_with_elementary_diff_level` (C0)

Proof. Using Lemma 8.5 we obtain a variable h such that $level(h) = level(a)$ and $b \in sub_{\mathrm{g}}(h)$. Clearly, two facts hold: (i) $a \neq b$, otherwise we would have a contradiction with $level(a) < level(b)$, and (ii) $a \neq h$, otherwise we would have a contradiction with $is\text{-}elem_{\mathrm{t}}(ty_{\mathrm{g}}(te, sc(mc), a))$ because g-variables of elementary types cannot have sub variables. From Lemma 8.8 we have $range_{\mathrm{g}}(te, ft, mc, a) \asymp range_{\mathrm{g}}(te, ft, mc, h)$. From Lemma 8.7 we have $range_{\mathrm{g}}(te, ft, mc, b) \subseteq range_{\mathrm{g}}(te, ft, mc, h)$. Ultimately, we conclude $range_{\mathrm{g}}(te, ft, mc, a) \asymp range_{\mathrm{g}}(te, ft, mc, b)$. □

Lemma 8.10 (Elementary variables are disjoint) Assume that a and b are different g-variables of elementary types, then their ranges are disjoint:

$$\begin{aligned} & a \neq b \ \wedge \ is\text{-}elem_{\mathrm{t}}(ty_{\mathrm{g}}(te, sc(mc), a)) \ \wedge \ is\text{-}elem_{\mathrm{t}}(ty_{\mathrm{g}}(te, sc(mc), b)) \\ & \longrightarrow \ range_{\mathrm{g}}(te, ft, mc, a) \asymp range_{\mathrm{g}}(te, ft, mc, b) \end{aligned}$$

Isabelle: `C0Acompilersimulation/abase_lemmata.elementary_impl_not_overlap_abase` (asm)
`C0Acompilersimulation/abase_lemmata.elementary_impl_not_overlap` (C0)

Proof. By case distinction on relation between the levels of variables.

Case $level(a) = level(b)$. By Lemma 8.8.

Case $level(a) \neq level(b)$. By Lemma 8.9.

□

The same is done for top local variables and heap variables. The proof for two variables from the different groups is as follows. In case of C0 machine there is nothing to prove since the variables belong to different memory frames. In the assembly machine we find a border address which separates these two memory frames and show that all addresses of the first memory frame are smaller than the border address and all addresses of the second memory frame are greater than the border address.

8.5 Correctness

Let c'_{C0} be a C0 configuration on which the effects of an assembly statement $\mathit{asm}(il)$ are projected, i.e., $\lfloor c'_{\mathrm{C0}} \rfloor = C0\text{-}asm\text{-}upd(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}}, c'_{\mathrm{ASM}}, alloc, gl)$. However, the compiler correctness relation does not necessarily hold between the C0 configuration c'_{C0} and the assembly configuration c'_{ASM}. The control consistency will be broken if the assembly statement is either (i) the last statement of a loop body, or (ii) the last statement of the *then* part of a conditional statement. The translation of these statements to the target code results in a list of assembly instructions il' which has to be executed by the assembly configuration c'_{ASM} in order to reach a consistent to c'_{C0} state. Note that il' contains only control instructions, and, hence does not affect any C0 variable. Executing il' we transit from c'_{ASM} to c''_{ASM} updating the program counters and regain consistency $consis(te, ft, c'_{\mathrm{C0}}, alloc, c''_{\mathrm{ASM}})$. This verification scenario is depicted in Figure 8.1. In the following we state this idea as a formal theorem.

Theorem 8.11 (Inline assembly preserves C0 consistency) Let te be a type environment and let ft be a function table. Let c_{ASM} and c_{C0} be assembly and C0 small-step semantics configurations before the execution of an inline assembly statement, and let

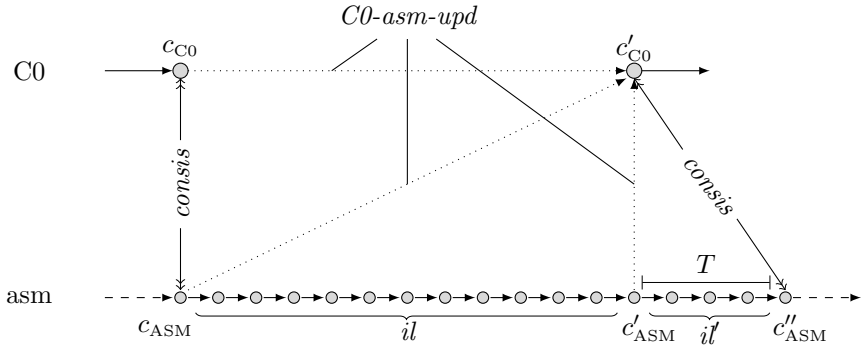

Figure 8.1: Switching C and assembly semantics.

c'_{ASM} and c'_{C0} be respective configurations after its execution. Let *alloc* be an allocation function and let *gl* be a list of g-variables affected by the inline assembly portion. Assume that (i) the assembly configuration c'_{ASM} is valid, (ii) the C0 configuration c_{C0} is valid, moreover the corresponding C0 program is translatable, (iii) c_{C0} is simulated by c_{ASM}, (iv) the next statement of c_{C0} is an inline-assembly statement, and (v) the projection of the inline-assembly effects to the C0 configuration is successful, then the assembly computation reaches in some number T of steps a configuration c''_{ASM}, such that (i) it is valid and simulates C0 configuration c'_{C0}, (ii) the memory, special-purpose and important general-purpose registers are unchanged in c''_{ASM} compared to c'_{ASM}, and (iii) for the last T steps it is guaranteed that the execution produces no interrupts on the ISA level:

$$
\begin{aligned}
& (te, ft, gst(c_{\mathrm{C0}}.mem)) \in xltbl_{\mathrm{prog}} \\
\wedge\ & consis(te, ft, c_{\mathrm{C0}}, alloc, c_{\mathrm{ASM}}) \\
\wedge\ & c_{\mathrm{C0}} \in C0\surd(te, ft) \\
\wedge\ & asm\surd(c'_{\mathrm{ASM}}) \\
\wedge\ & \textit{is-}\mathsf{asm}(stmt(c_{\mathrm{C0}})) \\
\wedge\ & \textit{C0-asm-upd}(te, ft, c_{\mathrm{C0}}, c_{\mathrm{ASM}}, c'_{\mathrm{ASM}}, gl) = \lfloor c'_{\mathrm{C0}} \rfloor \\
\longrightarrow\ & \exists\, T, c''_{\mathrm{ASM}} : \\
& \quad c''_{\mathrm{ASM}} = \delta^{T}_{\mathrm{ASM}}(c'_{\mathrm{ASM}}) \\
& \wedge\ consis(te, ft, c'_{\mathrm{C0}}, alloc, c''_{\mathrm{ASM}}) \\
& \wedge\ asm\surd(c''_{\mathrm{ASM}}) \\
& \wedge\ c''_{\mathrm{ASM}}.m = c'_{\mathrm{ASM}}.m \\
& \wedge\ c''_{\mathrm{ASM}}.spr = c'_{\mathrm{ASM}}.spr \\
& \wedge\ \forall\, r \in \{r_{\mathrm{sbase}}, r_{\mathrm{htop}}, r_{\mathrm{lframe}}, r_{\mathrm{jal}}\} :\ c''_{\mathrm{ASM}}.gpr[r] = c'_{\mathrm{ASM}}.gpr[r] \\
& \wedge\ asm\text{-}exec\text{-}props(c'_{\mathrm{ASM}}, \texttt{PROGBASE}, \\
& \qquad\qquad csize_{\mathrm{prog}}(te, gst(c_{\mathrm{C0}}.mem), ft), 0, 0, T)
\end{aligned}
$$

The instruction list il' described above contains only control instructions which do not affect the memory. Hence, in the predicate *asm-exec-props* we use two zeros at the place for the data range accessed during its execution.

Isabelle: `C0Acompilersimulation/asm_stmt_step_consistency.consistent_C0_conf_asm_update`

Proof. First, we prove all parts of the simulation relation between C0 configuration c'_{C0} and assembly configuration c'_{ASM} except for the control consistency (which, actually, does not always hold for c'_{C0} and c'_{ASM}). The code consistency $\mathit{consis}_{\mathrm{code}}(\mathit{te}, \mathit{ft}, c'_{\mathrm{C0}}, c'_{\mathrm{ASM}})$, clearly, holds because the program code range is unchanged. The arguments for the data consistency $\mathit{consis}_{\mathrm{d}}(\mathit{te}, \mathit{ft}, c'_{\mathrm{C0}}, \mathit{alloc}, c'_{\mathrm{ASM}})$ are as follows: we know that the update of the C0 configuration with the function $\mathit{C0\text{-}asm\text{-}upd}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, c_{\mathrm{ASM}}, c'_{\mathrm{ASM}}, \mathit{gl})$ succeeded, hence, the preconditions $\mathit{precond\text{-}C0\text{-}asm\text{-}upd}(\mathit{te}, \mathit{ft}, c_{\mathrm{C0}}, c_{\mathrm{ASM}}, c'_{\mathrm{ASM}}, \mathit{alloc}, \mathit{gl})$ were satisfied. We use them in order to conclude:

Consistency of frame headers. The additional frame information is stored at the place where no variables are stored.

Register consistency. Registers r_{sbase}, r_{htop}, and r_{lframe} are not changed as well as the structure of local stack, global, and heap memories. The set of all heap reachable variables could only be decreased.

Allocation consistency. The allocation function is not changed.

Value and pointer consistency. The most crucial point here is to show that all elementary variables occupy different C0 memory cells as well as different assembly-memory cells. We use Lemma 8.10.

Now we apply the correctness theorem for the control code [66, Theorem 10.4], which gives us all conclusions of our theorem, except for the code and data consistency terms of $\mathit{consis}(\mathit{te}, \mathit{ft}, c'_{\mathrm{C0}}, \mathit{alloc}, c''_{\mathrm{ASM}})$. Our remaining goal is to propagate $\mathit{consis}_{\mathrm{code}}(\mathit{te}, \mathit{ft}, c'_{\mathrm{C0}}, c'_{\mathrm{ASM}})$ and $\mathit{consis}_{\mathrm{d}}(\mathit{te}, \mathit{ft}, c'_{\mathrm{C0}}, \mathit{alloc}, c''_{\mathrm{ASM}})$ to hold with the assembly configuration c''_{ASM}. As we have just proven that the memory and important registers have the same value in c'_{ASM} and c''_{ASM}, the goal follows. □

Chapter 9

Verifying CVM Source Code

Contents

In this chapter we elaborate on the verification of the CVM implementation. In the frame of this work the following parts of the implementation have been formally verified: (i) context-switch routines [107], namely, process-context save and restore, (ii) the CVM's dispatcher, and (iii) three primitives [106], namely, `cvm_copy`(), `cvm_get_vm_word`(), and `cvm_set_vm_word`(). As for the primitives, we discuss only verification of `cvm_copy`() in this thesis because of two reasons. First, both remaining primitives are simple and all verification details that appear in their proofs also appear in the proof of the primitive for copying. Second, these primitives are excluded from the actual implementation state, because they are not used by the latest versions of the Verisoft's abstract kernels VAMOS and OLOS.

After a brief introduction in Section 9.1 of a number of auxiliary definitions common to all further sections devoted to verification of the source code we present in Section 9.2 an overall correctness proof of context switching and the CVM dispatcher. Section 9.3 discusses verification of CVM primitives on the example of the primitive `cvm_copy`().

9.1 Overview

We use the C0 small-step semantics extended with the inline assembly semantics for verification of the kernel's source code. We obtain the code formalization in the small-step semantics in Isabelle by means of a code translation tool developed in the frame of the Verisoft project. The tool also uniquely tags each statement with a numerical identifier. However, as we reason about the code of the concrete kernel obtained by means of the linking the translated CVM code with the code of the abstract kernel the statement identifiers are renumbered. Following the definition of the linking operator (cf. Section 6.2) in the obtained concrete kernel the statements with even identifiers will correspond to the CVM code.

Listing 9.1: Implementation of the initial jump.

```
1 j     (PROGBASE-4);
2 nop   ();
```

9.2 Process-Context Switch and Dispatching

In this section we focus on the verification of the process-context switch implementation in CVM. Context switching is closely related to some other portions of CVM like the initialization code and the CVM dispatcher. We elaborate on their correctness in the current section as well. Altogether, this section covers verification of the following pieces of code: (i) the initial jump to the beginning of the CVM code, (ii) the process-context save function `init_`() which also contains the code for initialization after reset, (iii) the process-context restore function `cvm_start`(), and (iv) the CVM dispatcher `dispatcher`(), a wrapper around the dispatcher of the abstract kernel, which is executed between context save and restore.

We start this section by discussing details of the context-switch implementation in CVM. Next we elaborate on its correctness. Further, in separate subsections we discuss verification of the CVM initialization code which is executed after a power up, correctness of process-context save, correctness of CVM's dispatcher, and verification of the process-context restore code.

9.2.1 Implementation

Initial Jump

Recall that VAMP JISR semantics set program counters to point to the zero address. However, the kernel code resides in the memory starting from address `PROGBASE`. In order to start execution of the kernel we need to reach the code of its first function by a jump to that address. Therefore, at the address zero we place the corresponding jump instruction (cf. Listing 9.1). An empty instruction is put after that because of the delayed branch mechanism.

Function `init_`()

Every run of a CVM-based kernel starts with the function `init_`() whose code is depicted in Listing 9.2. There are two possible cases of a kernel invocation: (i) after a computer power up or a reset signal, and (ii) on an interrupt occurring during execution of some user process or while the kernel is in the waiting for interrupts state. Depending on the case the function `init_`() either initializes the kernel memory structure with zeros or performs a process-context save. In the implementation this decision is taken by examining along lines 9–12 the least significant bit of the register *eca* which corresponds to the reset signal. In case this bit is on the *reset* part of the function (lines 13–27) is executed, otherwise we jump to line 28 and proceed with the *save* part. Both cases, however share saving of the general-purpose register one into the zero page in line 8.

Case *reset* (lines 13–27). Recall, that the C0 compiler reserves general-purpose registers 28, 29, and 30 for pointers to start of the global memory, heap memory, and the local stack, respectively. In lines 13–18 we initialize these registers with the constants $\texttt{ABASE}_{\text{lm}}$, $\texttt{ABASE}_{\text{hm}}$, and $\texttt{ABASE}_{\text{gm}}$. This defines the memory map of the kernel. Note that,

Listing 9.2: Kernel initialization and process-context save: function `init_()`.

```
1  int init_()
2  {
3   int          dummy;
4   unsigned int dispatcher_eca;
5   unsigned int dispatcher_edata;
6   unsigned int dispatcher_edpc;
7   assembler(
8    sw    (r1, r0, 8);
9    movs2i(r1, eca);
10   andi  (r1, r1, 1);
11   beqz  (r1, 64);
12   nop   ();
13   addi  (r30, r0, 448);
14   slli  (r30, r30, 12);
15   addi  (r29, r0, 1024 );
16   slli  (r29, r29, 12);
17   addi  (r28, r0, 384);
18   slli  (r28, r28, 12);
19   addi  (r2, r0, 64);
20   slli  (r2, r2, 12);
21   add   (r1, r0, r28);
22   sw    (r0, r1, 0);
23   subi  (r2, r2, 4);
24   bnez  (r2, -12);
25   addi  (r1, r1, 4);
26   beqz  (r0, 204);
27   nop   ();
28   sw    (r2, r0, 12);
29   sw    (r28, r0, 16);
30   addi  (r28, r0, 384);
31   slli  (r28, r28, 12);
32   addi  (r1, r28, asm_offset(cup));
33   lw    (r1, r1, 0);
34   slli  (r1, r1, 9);
35   addi  (r2, r28, asm_offset(pcb));
36   add   (r2, r2, r1);
37   sw    (r3, r2, 8);
38   sw    (r4, r2, 12);
39   sw    (r5, r2, 16);
40   sw    (r6, r2, 20);
41   sw    (r7, r2, 24);
42   sw    (r8, r2, 28);
43   sw    (r9, r2, 32);
44   sw    (r10, r2, 36);
45   sw    (r11, r2, 40);
46   sw    (r12, r2, 44);
47   sw    (r13, r2, 48);
48   sw    (r14, r2, 52);
49   sw    (r15, r2, 56);
50   sw    (r16, r2, 60);
51   sw    (r17, r2, 64);
52   sw    (r18, r2, 68);
53   sw    (r19, r2, 72);
54   sw    (r20, r2, 76);
55   sw    (r21, r2, 80);
56   sw    (r22, r2, 84);
57   sw    (r23, r2, 88);
58   sw    (r24, r2, 92);
59   sw    (r25, r2, 96);
60   sw    (r26, r2, 100);
61   sw    (r27, r2, 104);
62   sw    (r29, r2, 112);
63   sw    (r30, r2, 116);
64   sw    (r31, r2, 120);
65   lw    (r10, r0, 8);
66   sw    (r10, r2, 0);
67   lw    (r10, r0, 12);
68   sw    (r10, r2, 4);
69   lw    (r10, r0, 16);
70   sw    (r10, r2, 108);
71   movs2i(r7, epc);
72   sw    (r7, r2, 264);
73   movs2i(r7, edpc);
74   sw    (r7, r2, 268);
75   addi  (r30, r0, 448);
76   slli  (r30, r30, 12);
77   lw    (r29, r28, asm_offset(kheap));
78   movs2i(r10, eca);
79   sw    (r10, r30,
80          asm_offset(dispatcher_eca));
81   movs2i(r10, edpc);
82   sw    (r10, r30,
83          asm_offset(dispatcher_edpc));
84   movs2i(r10, edata);
85   sw    (r10, r30,
86          asm_offset(dispatcher_edata));
87  );
88  dummy =  dispatcher(dispatcher_eca,
89                      dispatcher_edata,
90                      dispatcher_edpc);
91  return 0;
92 }
```

these three constants are measured in pages — therefore, respective left shifts by 12 are done in the lines 14, 16, and 18. The subsequent lines fill the kernel's global memory with zeros because the global memory must be initialized. Lines 19–20 load the size of the global memory $\texttt{ABASE}_{\text{lm}} - \texttt{ABASE}_{\text{gm}}$ into a register while lines 21–25 implement a loop for the memory fill. The case ends with a jump to the line 78 at which some code which is shared between the *reset* and the *save* cases is placed.

Case *save* (lines 28–77). This part saves contents of visible hardware registers into the process control block of the interrupted user process. First, we save general-purpose register two into the zero page in line 28. This is done in order to have two registers — together with the one already saved in line 8 — available for storing temporary data during further computations. We save the content of register 28, a global memory pointer, into the zero page as well. Next, register 28 is set to value $\texttt{ABASE}_{\text{lm}} - \texttt{ABASE}_{\text{gm}}$ in lines 30-31. Along lines 28–36, we compute an offset in the array of process control

Listing 9.3: CVM elementary dispatcher: function `dispatcher()`.

```
1  int dispatcher
2         (unsigned int dispatcher_eca,
3          unsigned int dispatcher_edata,
4          unsigned int dispatcher_edpc)
5  {
6   int           dummy;
7   unsigned int pfh_res;
8   if ((dispatcher_eca & 1u) != 0u)
9   {
10   SR = 8254u;
11   dummy = pfh_init();
12  }
13  else if ((dispatcher_eca & 8u) != 0u)
14  {
15   pfh_res = pfh_touch_addr
16               (cup, dispatcher_edpc,
17                MM_SWAP_IN, 0u);
18   if (pfh_res != MM_INVALID_ADDR)
19   {
20    dispatcher_eca = 0u;
21   }
22  }
23  else if ((dispatcher_eca & 16u) != 0u)
24  {
25   pfh_res = pfh_touch_addr
26               (cup, dispatcher_edata,
27                MM_SWAP_IN, 0u);
28   if (pfh_res != MM_INVALID_ADDR)
29   {
30    dispatcher_eca = 0u;
31   }
32  };
33  if (dispatcher_eca != 0u)
34  {
35   cup = dispatcher_kernel
36            (dispatcher_eca,
37             dispatcher_edata);
38  };
39  if (cup > 0u && cup < PID_MAX)
40  {
41   dummy = cvm_start(cup);
42  }
43  else
44  {
45   dummy = cvm_wait();
46  };
47  return 0;
48 }
```

blocks corresponding to the interrupted process identified by `cup`. From line 37 to 64 we consecutively write the content of general-purpose registers 3 to 31, except for 28, directly into the PCB of process `cup`. In lines 65–70 we obtain the values of registers 1, 2, and 28 from the zero page and save them into the process control blocks. Lines 71–74 take care about special-purpose registers. We save the exception versions of program counters *epc* and *edpc*. Along lines 75–77 we assign the stack and heap pointers which reside in general-purpose registers 30 and 29 the values which correspond to the kernel. The stack pointer is set to $\texttt{ABASE}_{\text{lm}}$ while the heap pointer is assigned the value of the variable `kheap`. As our kernel support dynamic memory allocation this variable keeps track of the heap memory consumed by the kernel.

Shared code. The remaining lines of the assembly portion save the values of special-purpose registers *eca*, *edpc*, and *edata* — needed for interrupt handlers — into the local variables `dispatcher_eca`, `dispatcher_edpc`, and `dispatcher_edata`, respectively.

The function `init_`() finishes by invoking the CVM's dispatcher in line 88.

Function `dispatcher`()

The primary goal of the CVM dispatcher (cf. Listing 9.3) is to invoke the dispatcher of the abstract kernel which returns to CVM the identifier of the next scheduled process. This way, CVM knows the context of which process to restore next. Besides that the CVM dispatcher invokes, if needed, the page-fault handler initialization code and possibly calls the page-fault handler itself. The function `dispatcher`() has three arguments which are the stored values of registers *eca*, *edata*, and *edpc*. The implementation starts by a check along lines 8–12 of whether the kernel is invoked for the first time after a reset: the least significant bit of the variable `dispatcher_eca` is examined. If so, the status register variable `SR` is assigned the value of 8254 (cf. Section 5.2 for argumentation behind this number) and the initialization code of the page-fault handler is invoked. If we do not deal with a *reset* case we check whether any page-faults take place and try

Listing 9.4: Process-context restore: function `cvm_start()`.

```
1  int cvm_start(unsigned int pid)
2  {
3    int*   gpr1;
4    cup  = pid;
5    gpr1 = &(pcb[pid].
6             exception_frame[EF_GPR_1]);
7    assembler(
8      sw (r29, r28, asm_offset(kheap));
9    );
10   assembler(
11     lw      (r2, r28, asm_offset(SR));
12     movi2s (esr, r2);
13     lw      (r5, r30, asm_offset(gpr1));
14     add     (r1, r5, r0);
15     lw      (r2, r1, 0);
16     sw      (r2,r0, 8);
17     lw      (r2, r1, 4);
18     sw      (r2,r0, 12);
19     lw      (r3, r1, 8);
20     lw      (r4, r1, 12);
21     lw      (r5, r1, 16);
22     lw      (r6, r1, 20);
23     lw      (r7, r1, 24);
24     lw      (r8, r1, 28);
25     lw      (r9, r1, 32);
26     lw      (r10, r1, 36);
27     lw      (r11, r1, 40);
28     lw      (r12, r1, 44);
29     lw      (r13, r1, 48);
30     lw      (r14, r1, 52);
31     lw      (r15, r1, 56);
32     lw      (r16, r1, 60);
33     lw      (r17, r1, 64);
34     lw      (r18, r1, 68);
35     lw      (r19, r1, 72);
36     lw      (r20, r1, 76);
37     lw      (r21, r1, 80);
38     lw      (r22, r1, 84);
39     lw      (r23, r1, 88);
40     lw      (r24, r1, 92);
41     lw      (r25, r1, 96);
42     lw      (r26, r1, 100);
43     lw      (r27, r1, 104);
44     lw      (r28, r1, 108);
45     lw      (r29, r1, 112);
46     lw      (r30, r1, 116);
47     lw      (r31, r1, 120);
48     lw      (r2, r1, 264);
49     movi2s (epc, r2);
50     lw      (r2, r1, 268);
51     movi2s (edpc, r2);
52     lw      (r2, r1, 288);
53     movi2s (pto, r2);
54     lw      (r2, r1, 292);
55     movi2s (ptl, r2);
56     addi    (r2, r0, 1);
57     movi2s (emode, r2);
58     lw      (r1, r0, 8);
59     lw      (r2, r0, 12);
60     rfe     ();
61   );
62   return 0;
63 }
```

to handle them (lines 13–32). Note, that a page table length exception signaled by the `MM_INVALID_ADDR` return code of the page-fault handler is not handled within CVM but rather left in responsibility of the abstract kernel. If a page-fault was caused by other than PTL exception reasons and hence could be resolved, we will clear the variable `dispatcher_eca` and thus ignore the remaining interrupts. We can do so because only external interrupts may coexist with a page fault.This guarantees liveness for CVM. A page fault is a repeat-interrupt, i.e., the current user process did not make any progress. If the kernel would immediately be informed about the occurred external interrupts, rapid interrupt occurrences could starve the current user process. First, we check and handle a page fault on fetch. This is done at lines 13–22 by inspecting the fourth bit of the variable `dispatcher_eca` and a subsequent call to the handler. If there was no page fault on fetch we check along lines 23-32 for a page fault on load or store. Here the fifth bit is considered. If one of the page faults occurred and was handled we leave the dispatcher by restoring the interrupted process via a call to `cvm_start()` in line 41. Otherwise, we invoke the dispatcher of the abstract kernel in order to schedule the next user process (lines 33–38). In case some user process is scheduled we invoke it, otherwise we put CVM into an endless loop by invoking `cvm_wait()` (lines 39–46).

Function `cvm_start()`

The function for process-context restore (cf. Listing 9.4) is quite symmetrical to the *save* case of the function `init_()`. In the C0 portion of the function (lines 1–6) we compute an offset in the process control blocks array `pcb` corresponding to the next scheduled process `pid`. After the C0 part two inline assembly portions follow: this partition of the assembly code does not affect the generated object code but, however, eases verification.

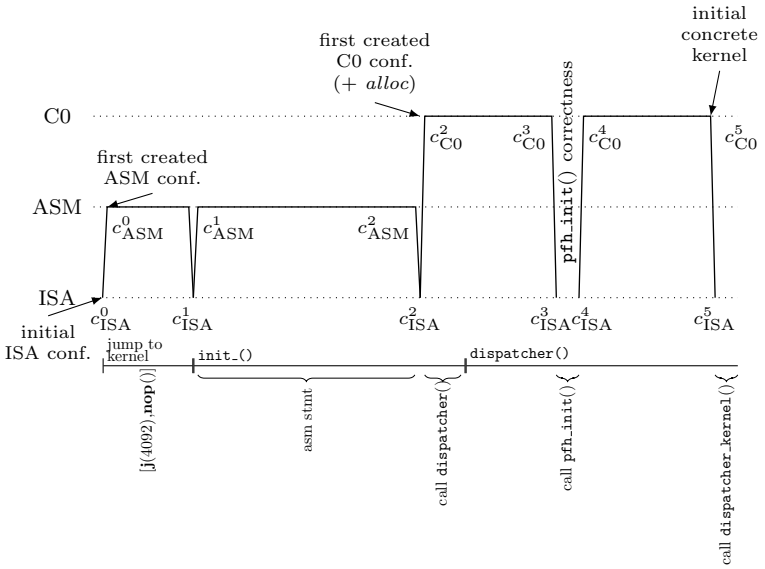

Figure 9.1: Verification diagram of the case *reset*.

The first assembly statement (lines 7–9) contains a single instruction which saves the value of the variable `kheap` storing the amount of the kernel heap memory into the general purpose register 29. In the second assembly statement we first save the value of the interrupt mask variable `SR` into the special purpose register *esr* (lines 11-12). Further, with lines 15–47 and 58–59 we load the values from the process control block of the process `pid` into general purpose registers. In the end of the function we load the values from PCB into such special registers as *epc*, *edpc*, *pto*, and *ptl*. Finally, we set the register *emode* on. The last instruction of the function is `rfe`. By this we change the hardware mode to user, leave the kernel, and start running the scheduled user process.

Further in this chapter, for each function name *fn* we will denote the formal definition of this function, i.e., an instance of the type *Func*, as *fn-proc* and its components as:

$$\begin{aligned} \textit{fn-proc.body} &= \textit{fn-body}, \\ \textit{fn-proc.params} &= \textit{fn-par}, \\ \textit{fn-proc.lvars} &= \textit{fn-loc}. \end{aligned}$$

For the statements we will omit the statement identifiers because they are generated by the code translation tool and depend on the complete code of the translated program but at the same time are irrelevant to the verification process.

9.2.2 Correctness of the Case *Reset*

We start discussing verification of the CVM implementation with a correctness proof of the case *reset*. Note that this will be the most detailed example covering typical peculiarities of formal reasoning about the CVM implementation like partitioning the code into logical portions depending on its semantical level or functionality, verifying code parts on the semantical level as abstractly as possible, and transferring correctness

results between the levels. For further cases and functions we will omit recurring details and concentrate on case-specific features.

The final correctness theorem for the case *reset* (cf. Theorem 9.10) covers not only the corresponding parts of the function `init_`(), but also the relevant portions of the CVM dispatcher. Altogether, we tie together in a single theorem correctness statements of (i) the following parts of the function `init_`(): pointers initialization, global memory initialization, coping values to the local variables, call of the `dispatcher`(), and (ii) the following parts of the function `dispatcher`(): initialization of the variable `SR`, call of the function `pfh_init`(), and the part of the CVM dispatcher until the call to the function `dispatcher_kernel`() passing by the page-fault handler calls.

Figure 9.1 sketches the verification process of the CVM-based kernel initialization. Such figures will be typical for this chapter. The figure depicts three semantical layers at which verification proceeds: C0, assembly, and ISA. At any point of time we try to achieve the desired verification result at the highest possible level. Finally, these results will be transfered to the ISA level by means of the C0-ISA ministack relation (cf. Section 4.3).

In Figure 9.1 the last pair of configurations $(c^5_{\mathrm{ISA}}, c^5_{\mathrm{C0}})$ are the states corresponding to the initial state of the CVM model. The whole correctness proof is divided into 5 parts. The first two parts — initial jump and the assembly portions from the function `init_`() — are verified on the assembly level. They are verified separately since the simulation theorem between VAMP ISA and VAMP assembly (cf. Theorem 4.8) assumes that the program code resides in a single continuous memory region, but in our case we use the memory between the jump code and kernel code to store some data. The third and the fifth parts (C0 statements between the assembly statement and the call to `pfh_init`(), and C0 statements between the `pfh_init`() call and the `dispatcher_kernel`() call) are verified on the C0 level. The fourth part is the call of the `pfh_init`() function. In this case we apply the correctness theorem of the page-fault handler initialization code [105].

By C^i_{ISA} we will denote the set of all configurations which share certain properties as the configuration c^i_{ISA}.

Initial Jump

We start by defining the set of ISA configurations C^0_{ISA} before execution of the initial jump. That a configuration c_{ISA} belongs to the set C^0_{ISA} means that c_{ISA} satisfies the preconditions to the initial jump code. Essential properties of a configuration $c_{\mathrm{ISA}} \in C^0_{\mathrm{ISA}}$ are that (i) the program counters point to the zero address, (ii) only the reset bit is set in the cause register, and (iii) the registers used for interrupt handling (*edata* and *edpc*) are set to zero. Formally:

$$
\begin{array}{ll}
 & \mathit{isa}\surd(c_{\mathrm{ISA}}) \\
\wedge & \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\
\wedge & \mathit{code\text{-}inv}(c_{\mathrm{ISA}}) \\
\wedge & \langle c_{\mathrm{ISA}}.\mathit{dpc}\rangle = 0 \\
\wedge & \langle c_{\mathrm{ISA}}.\mathit{pc}\rangle = 4 \\
\wedge & \langle c_{\mathrm{ISA}}.\mathit{spr}(\mathit{eca})\rangle = 1 \\
\wedge & \langle c_{\mathrm{ISA}}.\mathit{spr}(\mathit{edata})\rangle = 0 \\
\wedge & \langle c_{\mathrm{ISA}}.\mathit{spr}(\mathit{edpc})\rangle = 0 \\
\longrightarrow & c_{\mathrm{ISA}} \in C^0_{\mathrm{ISA}}.
\end{array}
$$

The postcondition of the initial jump is defined as a set of ISA configurations C^1_{ISA}. In case of reset the values of registers except for *eca*, *edata*, and *edpc* are not relevant to us. As for the memory we care only about the part where the code is stored. Ultimately, the postcondition states that a jump to the address `PROGBASE` is successfully executed and the program counters are updated correspondingly. Formally:

$$\begin{aligned}
& \mathit{isa}\surd(c_{\text{ISA}}) \\
\wedge\ & \mathit{is\text{-}sys\text{-}exec}_{\text{ISA}}(c_{\text{ISA}}) \\
\wedge\ & \mathit{code\text{-}inv}(c_{\text{ISA}}) \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{dpc}\rangle = \texttt{PROGBASE} \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{pc}\rangle = \texttt{PROGBASE} + 4 \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{spr}(\mathit{eca})\rangle = 1 \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{spr}(\mathit{edata})\rangle = 0 \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{spr}(\mathit{edpc})\rangle = 0 \\
\longrightarrow\ & c_{\text{ISA}} \in C^1_{\text{ISA}}.
\end{aligned}$$

As it follows from Figure 9.1 we verify the code of the initial jump in the assembly semantics. Because of that we also define the pre- and postconditions on the assembly level. We denote the sets of respective assembly configurations by C^0_{ASM} and C^1_{ASM}. However, we omit formal definitions of these configuration because they differ from C^0_{ISA} and C^1_{ISA} only in data representation.

Having specification of the initial jump formally defined we can formulate a correctness lemma for this portion of code. Since we formulate this lemma on the ISA level and will ease the proof by reasoning on the assembly level we need to introduce a conversion function from ISA configurations to assembly configurations. We obtain an assembly configuration from a given ISA configuration by means of the function

$$c_{\text{ISA}}2c_{\text{ASM}}(c_{\text{ISA}}) \stackrel{\text{def}}{=} c_{\text{ASM}},$$

where

$$\begin{aligned}
c_{\text{ASM}}.\mathit{dpc} &= \langle c_{\text{ISA}}.\mathit{dpc}\rangle, \\
c_{\text{ASM}}.\mathit{pc} &= \langle c_{\text{ISA}}.\mathit{pc}\rangle, \\
c_{\text{ASM}}.\mathit{gpr}[i] &= [c_{\text{ISA}}.\mathit{gpr}(\mathit{bin}(i))], \\
c_{\text{ASM}}.\mathit{spr}[i] &= [c_{\text{ISA}}.\mathit{spr}(\mathit{bin}(i))], \\
c_{\text{ASM}}.m(\mathit{ad}) &= [\mathit{read\text{-}isa}(c_{\text{ISA}}.m, 4 \cdot \mathit{ad})].
\end{aligned}$$

We justify correctness of this conversion function by the following two lemmas.

Lemma 9.1 (Correctness of conversion from ISA to assembly) The assembly configuration obtained from a valid ISA configuration by means of the function $c_{\text{ISA}}2c_{\text{ASM}}$ is equivalent to this ISA configuration:

$$\mathit{isa}\surd(c_{\text{ISA}}) \longrightarrow \mathit{isa\text{-}sim\text{-}asm}(c_{\text{ISA}}, c_{\text{ISA}}2c_{\text{ASM}}(c_{\text{ISA}})).$$

Isabelle: `cvm/additional/isa2asm_abs_lemmas.equiv_asm_isa_isa_to_asm`

Lemma 9.2 (Conversion from ISA to assembly preserves validity) The assembly configuration obtained from a valid ISA configuration by means of the function $c_{\mathrm{ISA}} 2 c_{\mathrm{ASM}}$ is valid:

$$\mathit{isa}\surd(c_{\mathrm{ISA}}) \longrightarrow \mathit{asm}\surd(c_{\mathrm{ISA}} 2 c_{\mathrm{ASM}}(c_{\mathrm{ISA}})).$$

Isabelle: `cvm/additional/isa2asm_abs_lemmas.is_dlx_conft_impl_is_ASMcore_isa_to_asm`

Now we prove correctness of the initial jump.

Lemma 9.3 (Correctness of initial jump) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) an execution sequence *seq* is well-formed, (iii) the hard disk properties hold, and (iv) an ISA processor configuration $c_{\mathrm{ISA+DS}}.\mathit{cpu}$ satisfies the precondition of the initial jump, then there exists a number of steps T whose execution brings ISA with devices into a configuration $c'_{\mathrm{ISA+DS}}$, such that (i) devices non-interference holds, (ii) the hard disk properties are preserved, and (iii) the ISA processor configuration $c'_{\mathrm{ISA+DS}}.\mathit{cpu}$ satisfies the postcondition of the initial jump:

$$\begin{array}{lll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) & \\
\wedge & \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
\wedge & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
\wedge & c_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^0_{\mathrm{ISA}} & \\
\longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}} : & \\
 & \quad \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} & \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev}(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) & \\
 & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
 & \wedge \quad c'_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^1_{\mathrm{ISA}}. &
\end{array}$$

Isabelle: `cvm/Reset/begin_jump_reset.begin_asm_code_correct_isa_reset`

Proof. In order to start verification on the assembly level we need an assembly configuration which can simulate the given ISA configuration. We obtain it from the c^0_{ISA} using the conversion function $c_{\mathrm{ISA}} 2 c_{\mathrm{ASM}}$. Lemma 9.2 allows us to apply assembly semantics. Using the assembly semantics for the execution of the two instructions constituting the code of the initial jump and applying simulation theorems we conclude the properties of c^1_{ISA}.

□

Assembly Statement of the Function `init_`()

The assembly portion of the initialization/context save function which belongs to the case *reset* implements the following changes: (i) the program counters are set to the end address of the assembly portion, (ii) the global memory, local memory, and heap

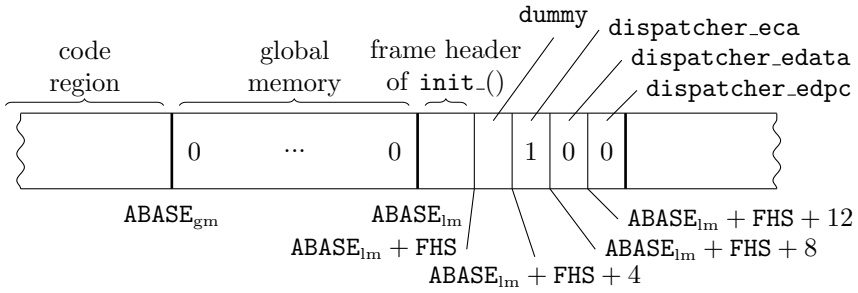

Figure 9.2: Memory structure during execution of the function `init_()` in the case *reset*.

pointers are loaded, (iii) the memory region from $\texttt{ABASE}_{\text{gm}}$ to $\texttt{ABASE}_{\text{lm}}$ is filled with zeros, (iv) the values of the special purpose registers are written to the local variable. The allocated addresses of these variables are $\texttt{ABASE}_{\text{lm}} + \texttt{FHS} + 4$, $\texttt{ABASE}_{\text{lm}} + \texttt{FHS} + 8$, and $\texttt{ABASE}_{\text{lm}} + \texttt{FHS} + 12$ (cf. Figure 9.2). `FHS` bytes are reserved for some information of the local frame; 4, 8, and 12 are displacements within the memory frame. We denote the assembly code from the first statement of the function `init_()` by $\pi^{\text{init}}_{\text{ASM}}$. The described effect of this portion of code are formally stated in the following postcondition, defined as a set of ISA configurations C^2_{ISA}:

$$
\begin{aligned}
& \mathit{isa}\surd(c_{\text{ISA}}) \\
\wedge\ & \mathit{is\text{-}sys\text{-}exec}_{\text{ISA}}(c_{\text{ISA}}) \\
\wedge\ & \mathit{code\text{-}inv}(c_{\text{ISA}}) \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{dpc}\rangle = \texttt{PROGBASE} + |\pi^{\text{init}}_{\text{ASM}}| \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{pc}\rangle = \texttt{PROGBASE} + |\pi^{\text{init}}_{\text{ASM}}| + 4 \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{gpr}(\mathit{bin}(28))\rangle = \texttt{ABASE}_{\text{gm}} \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{gpr}(\mathit{bin}(29))\rangle = \texttt{ABASE}_{\text{hm}} \\
\wedge\ & \langle c_{\text{ISA}}.\mathit{gpr}(\mathit{bin}(30))\rangle = \texttt{ABASE}_{\text{lm}} \\
\wedge\ & \mathit{get\text{-}data}_{\text{ISA}}(c_{\text{ISA}}, \texttt{ABASE}_{\text{gm}}, (\texttt{ABASE}_{\text{lm}} - \texttt{ABASE}_{\text{gm}})/4) = 0^{(\texttt{ABASE}_{\text{lm}} - \texttt{ABASE}_{\text{gm}})/4} \\
\wedge\ & \mathit{get\text{-}data}_{\text{ISA}}(c_{\text{ISA}}, \texttt{ABASE}_{\text{lm}} + \texttt{FHS} + 4, 3) = [1,0,0] \\
\longrightarrow\ & c_{\text{ISA}} \in C^2_{\text{ISA}}.
\end{aligned}
$$

Correctness of the part is stated in the following lemma.

Lemma 9.4 (Correctness of assembly statement in `init_()`) Assume that (i) the properties of the abstract kernel hold fo π_{AK}, (ii) an execution sequence *seq* is well-formed, (iii) the hard disk properties hold, and (iv) an ISA processor configuration $c_{\text{ISA+DS}}.\mathit{cpu}$ satisfies the postcondtion of the initial jump, then there exists a number of steps T whose execution brings ISA with devices into a configuration $c'_{\text{ISA+DS}}$, such that (i) devices non-interference holds, (ii) the hard disk properties are preserved, and (iii) the ISA processor configuration $c'_{\text{ISA+DS}}.\mathit{cpu}$ satisfies the postcondition of the as-

sembly statement of the function `init_`():

$$
\begin{array}{ll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \\
\wedge & \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & c_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^1_{\mathrm{ISA}} \\
\longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}} : \\
 & \quad \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev}(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & \wedge \quad c'_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^2_{\mathrm{ISA}}.
\end{array}
$$

Isabelle: `cvm/Reset/init_asm_reset.init_code_reset_correct_isa`

Note that the code to be verified contains also a loop at lines 22–25. The loop is executed $\langle c_{\mathrm{ISA}}.\mathit{gpr}(\mathit{bin}(2))\rangle/4$ times.

C0 Statements of `init_`() and `dispatcher`() before the `pfh_init`() Call

We execute and verify this part on the C0 level. During this we do not care much about the properties on the ISA level except for the following conditions of valid execution: (i) the ISA configuration is valid, (ii) the system mode is on, and (iii) the code invariant holds. In conjunction these three properties we define the postcondition to the currently discussed portion of code on the ISA level as a set of configurations C^3_{ISA}:

$$
\begin{array}{ll}
 & \mathit{isa}\surd(c_{\mathrm{ISA}}) \\
\wedge & \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\
\wedge & \mathit{code\text{-}inv}(c_{\mathrm{ISA}}) \\
\longrightarrow & c_{\mathrm{ISA}} \in C^3_{\mathrm{ISA}}.
\end{array}
$$

Now we define the postcondition on the C0 level. Before that let us introduce the following notation for a variable g, a memory configuration mc, and a memory cell value v:

$$\Vert\!\downarrow g \downarrow\!\Vert_{mc} = v$$

which combines two properties: the variable g is initialized with respect to the memory configuration mem and has the value v:

$$
\begin{array}{ll}
 & \mathit{initialized}_{\mathrm{g}}(mc, g) \\
\wedge & \mathit{value}_{\mathrm{g}}(mc, g) = v.
\end{array}
$$

Further, at places where it is clear which memory configuration is used we will omit it.

The C0 postcondition is defined as a set of C0 configurations C^3_{C0}. Essentially, it is a conjunction of the following facts: (i) the C0 configuration is valid and the invariant on the global memory structure holds, (ii) the very first global variable (`SR`) has the value of 8254 and all other global variables have predefined initial values, (iii) the heap memory is empty, (iv) the local memory stack has two frames corresponding to the functions `init_`() and `dispatcher`(), (v) the variable storing values of registers *eca*, *edata*, and *edpc* has the values 1, 0, and 0, respectively, and (vi) the program rest is equal the

the predefined constant $\texttt{prog}^3$, which encodes the body of the function `dispatcher`() starting from line 11 in Listing 9.3. Formally:

$$
\begin{array}{ll}
 & c_{\mathrm{C0}} \in C0''\surd(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\
\wedge & \mathit{structure}^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge & c_{\mathrm{C0}}.\mathit{mem.gm.ct}(i) = \begin{cases} \mathit{nat}(8254) & \text{if } i = 0 \\ \mathit{init}_{\mathrm{ct}}(\mathit{gst}_{\mathrm{CK}}(\pi_{\mathrm{AK}}))(i) & \text{otherwise} \end{cases} \\
\wedge & c_{\mathrm{C0}}.\mathit{mem.hm} = \mathit{init}_{\mathrm{mem}}([]) \\
\wedge & |c_{\mathrm{C0}}.\mathit{mem.lm}| = 2 \\
\wedge & c_{\mathrm{C0}}.\mathit{mem.lm}[\![0]\!].\mathit{mfr.st} = \mathit{st}_{\mathrm{fun}}(\mathit{init\text{-}proc}) \\
\wedge & c_{\mathrm{C0}}.\mathit{mem.lm}[\![1]\!].\mathit{mfr.st} = \mathit{st}_{\mathrm{fun}}(\mathit{dispatcher\text{-}proc}) \\
\wedge & \Vert\, \mathit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_eca}) \,\Vert = \mathit{nat}(1) \\
\wedge & \Vert\, \mathit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_edata}) \,\Vert = \mathit{nat}(0) \\
\wedge & \Vert\, \mathit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_edpc}) \,\Vert = \mathit{nat}(0) \\
\wedge & c_{\mathrm{C0}}.\mathit{prog} = \texttt{prog}^3 \\
\longrightarrow & c_{\mathrm{C0}} \in C^3_{\mathrm{C0}}.
\end{array}
$$

Next, we state and prove the correctness lemma for the current code portion.

Lemma 9.5 (Correctness of C0 parts of `init_`() and `dispatcher`()) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) an execution sequence *seq* is well-formed, (iii) the hard disk properties hold, and (iv) an ISA processor configuration $c_{\mathrm{ISA+DS}}.\mathit{cpu}$ satisfies the postcondition of the assembly statement of the function `init_`(), then there exists a number of steps T whose execution brings ISA with devices into a configuration $c'_{\mathrm{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) devices non-interference holds, (ii) the hard disk properties are preserved, (iii) the C0 configuration c'_{C0} is simulated by the ISA configuration $c'_{\mathrm{ISA+DS}}.\mathit{cpu}$, and (iv) the postconditions on the ISA and C0 levels hold:

$$
\begin{array}{ll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \\
\wedge & \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & c_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^2_{\mathrm{ISA}} \\
\longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}} : \\
 & \quad \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev}(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & \wedge \quad \mathit{C0\text{-}sim\text{-}isa}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
 & \wedge \quad c'_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^3_{\mathrm{ISA}} \\
 & \wedge \quad c'_{\mathrm{C0}} \in C^3_{\mathrm{C0}}.
\end{array}
$$

Isabelle: `cvm/Reset/init_dispatcher_reset.init_dispatcher_reset_isa_correct`

Proof. The lemma is proven by applying the C0 small-step semantics. In order to proceed with verification on the C0 level we first of all need to find an initial C0 configuration

consistent to the ISA configuration c^2_{ISA}. We denote this configuration by c^{init}_{C0} and define it as follows:

$$
\begin{aligned}
c^{init}_{\text{C0}}.prog &= \mathit{sCall}(\texttt{dummy}, \texttt{dispatcher}, \\
&\quad [\mathit{var}(\texttt{dispatcher\_eca}, \mathit{unsgnd}_{\text{T}}), \\
&\quad \mathit{var}(\texttt{dispatcher\_edata}, \mathit{unsgnd}_{\text{T}}), \\
&\quad \mathit{var}(\texttt{dispatcher\_edpc}, \mathit{unsgnd}_{\text{T}})]), \\
c^{init}_{\text{C0}}.mem.gm.st &= gst_{\text{CK}}(\pi_{\text{AK}}.st), \\
c^{init}_{\text{C0}}.mem.gm.init &= vns(gst_{\text{CK}}(\pi_{\text{AK}}.st)), \\
c^{init}_{\text{C0}}.mem.gm.ct &= init_{\text{ct}}(gst_{\text{CK}}(\pi_{\text{AK}})), \\
c^{init}_{\text{C0}}.mem.hm &= init_{\text{mem}}([]), \\
c^{init}_{\text{C0}}.mem.lm &= [(\mathit{lm\text{-}frame}^{init}, \mathsf{A})], \\
\mathit{lm\text{-}frame}^{init}.st &= st_{\text{fun}}(\mathit{init\text{-}proc}), \\
\mathit{lm\text{-}frame}^{init}.ct &= vns(st_{\text{fun}}(\mathit{init\text{-}proc})), \\
\mathit{lm\text{-}frame}^{init}.init &= \mathit{lm\text{-}content}^{init}, \\
\mathit{lm\text{-}content}^{init}(i) &= \begin{cases} \mathit{nat}(1) & \text{if } i = 1 \\ \mathit{nat}(0) & \text{if } i = 2 \\ \mathit{nat}(0) & \text{if } i = 3 \\ \mathsf{A} & \text{otherwise} \end{cases} .
\end{aligned}
$$

Note, that in the last equation the content of the local memory frame is defined according to the symbol table of the function `init_`():

$$
\begin{aligned}
1 &= ba_{\text{v}}(st_{\text{fun}}(\mathit{init\text{-}proc}), \texttt{dispatcher\_eca}), \\
2 &= ba_{\text{v}}(st_{\text{fun}}(\mathit{init\text{-}proc}), \texttt{dispatcher\_edata}), \\
3 &= ba_{\text{v}}(st_{\text{fun}}(\mathit{init\text{-}proc}), \texttt{dispatcher\_edpc}).
\end{aligned}
$$

We use Lemma 9.6 and Lemma 9.7 stated and proven below to justify that the C0 configuration c^{init}_{C0} satisfies the postcondition C^2_{ISA}. □

Lemma 9.6 (Construction correctness of $c^{init}_{\mathbf{C0}}$) Assume that the abstract kernel properties hold for π_{AK} and the ISA code invariants are satisfied for a valid configuration c_{ISA} which encodes the postcondition C^2_{ISA}, then c_{ISA} simulates the C0 configuration c^{init}_{C0}:

$$
\begin{aligned}
& \mathit{abs\text{-}kernel\text{-}props}(\pi_{\text{AK}}) \\
\wedge\ & \mathit{isa}_{\surd}(c_{\text{ISA}}) \\
\wedge\ & \mathit{code\text{-}inv}(c_{\text{ISA}}) \\
\wedge\ & c_{\text{ISA}} \in C^2_{\text{ISA}} \\
\longrightarrow\ & \mathit{C0\text{-}sim\text{-}isa}(te_{\text{CK}}(\pi_{\text{AK}}), ft_{\text{CK}}(\pi_{\text{AK}}), c^{init}_{\text{C0}}, c_{\text{ISA}}).
\end{aligned}
$$

Isabelle: `cvm/Reset/init_consis_reset.init_code_reset_isa_post_impl_consistent_init_cvm_c0_config_isa_to_asm`

Proof. We start the proof by unfolding the C0-ISA simulation relation *C0-sim-isa*. According to its definition we need to instantiate the intermediate assembly configuration and the C0 allocation function. As an assembly machine we use configuration

$c_{\mathrm{ASM}} = c_{\mathrm{ISA}} 2 c_{\mathrm{ASM}}(c_{\mathrm{ISA}})$. Using Lemma 9.2 and Lemma 9.1 we conclude the validity of this assembly configuration $asm\surd(c_{\mathrm{ASM}})$ as well as its equivalence to the original ISA machine $isa\text{-}sim\text{-}asmc_{\mathrm{ISA}} c_{\mathrm{ASM}}$. As for the allocation function we instantiate it with the value $alloc^{init}$ defined below using $sc = sc(ak_{\mathrm{init}}(\pi_{\mathrm{AK}}).mem)$:

$$alloc^{init}(g) \stackrel{\mathrm{def}}{=} \begin{cases} (\mathtt{ABASE}_{\mathrm{lm}} + \mathtt{FHS}, 4) & \text{if } g = gvar_{\mathrm{lm}}(0, \mathtt{dummy}) \\ (\mathtt{ABASE}_{\mathrm{lm}} + \mathtt{FHS} + 4, 4) & \text{if } g = gvar_{\mathrm{lm}}(0, \mathtt{dispatcher\_eca}) \\ (\mathtt{ABASE}_{\mathrm{lm}} + \mathtt{FHS} + 8, 4) & \text{if } g = gvar_{\mathrm{lm}}(0, \mathtt{dispatcher\_edata}) \\ (\mathtt{ABASE}_{\mathrm{lm}} + \mathtt{FHS} + 12, 4) & \text{if } g = gvar_{\mathrm{lm}}(0, \mathtt{dispatcher\_edpc}) \\ alloc^{init}_{\mathrm{gm}} & \text{if } \; is\text{-}gm\text{-}gvar(g) \\ & \quad \wedge \; (te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), sc, g) \in gvars\surd \\ \mathsf{A} & \text{otherwise} \end{cases},$$

where

$$alloc^{init}_{\mathrm{gm}} = (abase_{\mathrm{g}}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}), sc, g), asize_{\mathrm{t}}(ty_{\mathrm{g}}(sc, g))).$$

The values of the allocation function for the local variables are chosen according to the memory structure depicted in Figure 9.2.

The remaining goal to be proven is

$$consis(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c^{init}_{\mathrm{C0}}, alloc^{init}, c_{\mathrm{ISA}} 2 c_{\mathrm{ASM}}(c_{\mathrm{ISA}})).$$

Subgoal 1. Code consistency $consis_{\mathrm{code}}$. It follows directly from $code\text{-}inv(c_{\mathrm{ISA}})$.

Subgoal 2. Control consistency $consis_{\mathrm{c}}$. Since the dispatcher call statement follows directly after the assembly statement, it is not difficult to show that the start address of the call statement is the same as the end address of the assembly statement and equals to $\mathtt{PROGBASE} + |\pi^{\mathrm{init}}_{\mathrm{ASM}}|$. The recursion depth of the local stack is equal to one. Since there are no frames created by the call nothing has to be proven about return addresses of stack frames.

Subgoal 3. Register consistency $consis_{\mathrm{r}}$. We exploit the following facts:

$$\begin{aligned} \langle c_{\mathrm{ISA}}.gpr(bin(28)) \rangle &= \mathtt{ABASE}_{\mathrm{gm}}, \\ \langle c_{\mathrm{ISA}}.gpr(bin(29)) \rangle &= \mathtt{ABASE}_{\mathrm{hm}}, \\ \langle c_{\mathrm{ISA}}.gpr(bin(30)) \rangle &= \mathtt{ABASE}_{\mathrm{lm}}. \end{aligned}$$

All we need to show is:

$$\begin{aligned} &abase_{\mathrm{gm}}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}), gst_{\mathrm{CK}}(\pi_{\mathrm{AK}})) = \mathtt{ABASE}_{\mathrm{gm}}, \\ &\mathtt{ABASE}_{\mathrm{hm}} + asize_{\mathrm{heap}}([]) = \mathtt{ABASE}_{\mathrm{hm}}, \\ &abase_{\mathrm{lm}}(te, ft, sc, 0) = \mathtt{ABASE}_{\mathrm{lm}}, \end{aligned}$$

which follows from the abstract kernel properties $abs\text{-}kernel\text{-}props(\pi_{\mathrm{AK}})$.

Subgoal 4. Frame header consistency $consis_{\mathrm{fh}}$. It is trivial because the recursion depth of the local stack is equal to one.

We reason about the allocation, value, and pointer consistency statements having the following fact in mind:

$$\begin{aligned}\forall\, g:\quad & (te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), sc, g) \in gvars\surd \\ \longrightarrow\quad & \textit{is-gm-gvar}(g) \\ \vee\quad & g = \textit{gvar}_{\mathrm{lm}}(0, \texttt{dummy}) \\ \vee\quad & g = \textit{gvar}_{\mathrm{lm}}(0, \texttt{dispatcher\_eca}) \\ \vee\quad & g = \textit{gvar}_{\mathrm{lm}}(0, \texttt{dispatcher\_edata}) \\ \vee\quad & g = \textit{gvar}_{\mathrm{lm}}(0, \texttt{dispatcher\_edpc}),\end{aligned}$$

as well as that there are not heap variables.

Subgoal 5. Allocation consistency $\textit{consis}_{\mathrm{alloc}}$. For global and local variables it follows directly from the definition of $\textit{alloc}^{\textit{init}}$.

Subgoal 6. Value consistency $\textit{consis}_{\mathrm{v}}$. Non-pointer global variables are initialized and have one of the following values: $\textit{bool}(\mathsf{F})$, $\textit{int}(0)$, $\textit{char}(0)$, or $\textit{nat}(0)$. They all correspond to the zero constant in the assembly machine which follows from $\textit{get-data}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, \texttt{ABASE}_{\mathrm{gm}}, (\texttt{ABASE}_{\mathrm{lm}} - \texttt{ABASE}_{\mathrm{gm}})/4) = 0^{(\texttt{ABASE}_{\mathrm{lm}} - \texttt{ABASE}_{\mathrm{gm}})/4}$. Three initialized local variables have values 1, 0, and 0. From $\textit{get-data}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, \texttt{ABASE}_{\mathrm{lm}} + \texttt{FHS} + 4, 3) = [1, 0, 0]$ and the definition of $\textit{alloc}^{\textit{init}}$ we conclude that the values match.

Subgoal 7. Pointer consistency $\textit{consis}_{\mathrm{p}}$. We need to show it only for global pointers because there are no local pointer in the source code. Similarly to the value consistency, the initialized value for a global pointer is $\textit{ptr}(\bot)$, which also corresponds to the zero.

□

Lemma 9.7 (Validity of $c^{\textit{init}}_{\mathbf{C0}}$) Assume that the abstract kernel properties hold, then the configuration $c^{\textit{init}}_{\mathrm{C0}}$ is valid:

$$\textit{abs-kernel-props}(\pi_{\mathrm{AK}}) \;\longrightarrow\; c^{\textit{init}}_{\mathrm{C0}} \in C0''\surd(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}})).$$

Isabelle: `cvm/Reset/init_valid_conf_reset.init_cvm_c0_config_in_valid_confs`

Proof. All properties about the type environment and the function table follow directly from $\textit{abs-kernel-props}(\pi_{\mathrm{AK}})$. Properties about the memory configuration and the program rest are proven by unfolding respective definitions. □

Correctness of the Call to `pfh_init`()

Correctness of the initialization code of the page-fault handler was formally proven by Starostin [105]. The distinctive feature of this code portion is that it establishes some crucial CVM correctness criteria like the $\mathcal{B}$ relation for the first time during some considered CVM run. Below, we briefly overview some parts of the specification and state the correctness lemma for this case.

Preconditions on the C0 level state such natural things as (i) the validity of the C0 configuration, (ii) the C0 machine is calling the function `pfh_init`() and writes its result

back to the variable `dummy` which belongs to the lop-local memory frame, (iii) the global variables of the C0 machine coincide with those of the concrete kernel and are initialized, (iv) the heap memory is empty, (v) the local memory is appropriately bounded, and (vi) the global variable `SR` is initialized with the number 8254:

$$\begin{array}{lcl} \textit{pfh-init-PRE}_{\mathrm{C0}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) & \stackrel{\mathrm{def}}{=} & c_{\mathrm{C0}} \in C0'' \surd (te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\ & \wedge & \textit{stmt}(c_{\mathrm{C0}}.\textit{prog}) = \textit{sCall}(\mathtt{dummy}, \mathtt{pfh\_init}, [], \textit{sid}) \\ & \wedge & \mathtt{dummy} \in \textit{vns}(\textit{lst}_{\mathrm{top}}(c_{\mathrm{C0}}.\textit{mem})) \\ & \wedge & \textit{gst}(c_{\mathrm{C0}}.\textit{mem}) = \textit{gst}_{\mathrm{CK}}(\pi_{\mathrm{AK}}) \\ & \wedge & \textit{vns}(\textit{gst}_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \subseteq c_{\mathrm{C0}}.\textit{mem.gm.init} \\ & \wedge & \textit{hst}(c_{\mathrm{C0}}.\textit{mem}) = [] \\ & \wedge & \textit{asize}_{\mathrm{st}^*}(\textit{sc}(c_{\mathrm{C0}}.\textit{mem}).\textit{lst}) + 52 \leq \mathtt{ABASE}_{\mathrm{hm}} - \mathtt{ABASE}_{\mathrm{lm}} \\ & \wedge & \| \textit{gvar}_{\mathrm{gm}}(\mathtt{SR}) \| = \textit{nat}(8254). \end{array}$$

Here we use the function $\textit{asize}_{\mathrm{st}^*}$ to compute the size of the local memory stack. This function is defined as the sum of symbol table sizes:

$$\textit{asize}_{\mathrm{st}^*}(\textit{lsts}) \stackrel{\mathrm{def}}{=} \sum_{i=0}^{i<|\textit{lsts}|} \textit{asize}_{\mathrm{st}}(\textit{lsts}[i]).$$

The number 52 comes from the estimation of the local stack size for the `pfh_init`() execution:

$$(ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathtt{pfh\_init}, 52) \in \textit{SE}.$$

Additionally, the predicate $\textit{pfh-init-PRE}_{\mathrm{PFH}}(te, mc)$ is defined stating the precondition about the page-fault handler data structures, namely: the reverse lookup array `ppx2pd`, the stack of free big pages `bpfree_stack`, the big-page table space `bptspace`, and the (relevant to the page-fault handler) process control blocks `pcb`. The precondition states that all these data structures are initialized with zeros.

The C0 postcondition of a call to the page-fault handler initialization code, basically, states that the call was successfully processed: the validity of the C0 configuration is preserved, the call statement was removed from the program rest, and all variables except those touched by the code remain unchanged. Moreover, the C0 memory invariant for the function call execution holds:

$$\begin{array}{lcl} \textit{pfh-init-POST}_{\mathrm{C0}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) & \stackrel{\mathrm{def}}{=} & c'_{\mathrm{C0}} \in C0'' \surd (te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\ & \wedge & c'_{\mathrm{C0}}.\textit{prog} = \textit{rem-1st-stmt}(c_{\mathrm{C0}}.\textit{prog}) \\ & \wedge & \textit{unchanged}_{\mathrm{GM,HM}}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\textit{mem}, c'_{\mathrm{C0}}.\textit{mem}, \\ & & \qquad \textit{pfh-gm-vars}, \textit{pfh-hm-ind}) \\ & \wedge & \textit{is-C0mem-inv}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\textit{mem}, c'_{\mathrm{C0}}.\textit{mem}, \\ & & \qquad \textit{hst}_{\mathrm{CVM}}, \mathtt{dummy}, \textit{int}(0)). \end{array}$$

The postcondition over the page-fault handler data structures are, essentially, the simulation relation for user processes $\mathcal{B}$, the page-fault handler invariant *pfh-inv*, and the zero-filled page condition *zfp-cond*. All of them appear directly in the correctness lemma of the page-fault handler initialization code.

Note, that page-fault handler execution may touch the hard disk. Therefore, we can claim that the processor does not interfere with all devices except for the hard

disk. We express this formally with the predicate *non-interf-dev′* which differs from its unprimed version (cf. Definition 3.19) in that it excludes the hard-disk from the universal quantification of (non-interfered) devices.

Lemma 9.8 (Correctness of `pfh_init()`) Let $c_{\mathrm{ISA+DS}}$ be a configuration of VAMP ISA with devices, let *seq* be an ISA execution sequence, and let c_{C0} be a C0 configuration of the concrete kernel. Assume that (i) the abstract kernel properties holds for π_{AK}, (ii) the execution sequence *seq* and the hard disk are valid, (iii) the ISA processor configuration $c_{\mathrm{ISA+DS}}.cpu$ is valid and simulates the C0 configuration c_{C0}, (iv) the ISA code invariant is satisfied, (v) the ISA machine runs in the system mode, and (vi) both C0 precondition and the preconditions for the page-fault handler data structures are satisfied, then there exists a number of steps T during whose execution the ISA with devices transits to a configuration $c'_{\mathrm{ISA+DS}}$, and a C0 configuration c'_{C0} such that the following holds: (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk remains valid, (iii) the ministack simulation relation holds between $c'_{\mathrm{ISA+DS}}.cpu$ and c'_{C0}, (iv) the ISA code invariant is preserved and ISA remains in the system mode, (v) the zero filled page condition is established, (vi) the C0 postcondition holds, (vii) the page-fault handler invariant holds, and (viii) the simulation relation for user processes is established:

$$
\begin{array}{lll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) & \\
\wedge & \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
\wedge & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
\wedge & \mathit{C0\text{-}sim\text{-}isa}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
\wedge & \mathit{isa}\surd(c_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
\wedge & \mathit{code\text{-}inv}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
\wedge & \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ISA}}(c_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
\wedge & \mathit{pfh\text{-}init\text{-}PRE}_{\mathrm{C0}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) & \\
\wedge & \mathit{pfh\text{-}init\text{-}PRE}_{\mathrm{PFH}}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\mathit{mem}) & \\
\longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}} : & \\
 & \quad \delta^{T}_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} & \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev'}(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) & \\
 & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) & \\
 & \wedge \quad \mathit{C0\text{-}sim\text{-}isa}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
 & \wedge \quad \mathit{isa}\surd(c'_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
 & \wedge \quad \mathit{code\text{-}inv}(\pi_{\mathrm{AK}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
 & \wedge \quad \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ISA}}(c'_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
 & \wedge \quad \mathit{zfp\text{-}cond}(c'_{\mathrm{ISA+DS}}.\mathit{cpu}) & \\
 & \wedge \quad \mathit{pfh\text{-}init\text{-}POST}_{\mathrm{C0}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) & \\
 & \wedge \quad \mathit{pfh\text{-}inv}(c^{\mathrm{init}}_{\mathrm{PFH}}, \mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c'_{\mathrm{C0}}.\mathit{mem}) & \\
 & \wedge \quad \mathcal{B}(\mathit{ups}_{\mathrm{init}}, c'_{\mathrm{ISA+DS}}). &
\end{array}
$$

Isabelle: `pfh/NoXCall/pfhInitTopLevel.pfh_init_correct`

C0 Statements of `dispatcher`() between the Calls to `pfh_init`() and `dispatcher_kernel`()

The remaining portion of the code for the case *reset* is the easiest. Only a single control statement is executed: the conditional statement at line 33 of Listing 9.3. The statement is supposed to call the dispatcher of the abstract kernel in case the stored value of the *eca* register differs from zero. Since we are dealing with the *reset* case the corresponding bit is raised in the exceptional cause register and, hence, a call to the abstract kernel dispatcher takes place.

The precondition to this code part on the C0 level, defined as a set of C0 configurations C^4_{C0}, comprises such facts about the C0 configuration as its validity, sufficiency of the heap memory, invariants over the global and heap memory parts of the concrete kernel, information about the local stack, as well as the state of the program rest:

$$
\begin{array}{ll}
 & c_{\mathrm{C0}} \in C0''\surd(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\
\wedge & avail_{\mathrm{heap}}(c_{\mathrm{C0}}) \\
\wedge & structure^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge & structure^{\mathrm{ck}}_{\mathrm{HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge & |c_{\mathrm{C0}}.mem.lm| = 2 \\
\wedge & c_{\mathrm{C0}}.mem.lm[\![0]\!].mfr.st = st_{\mathrm{fun}}(\textit{init-proc}) \\
\wedge & c_{\mathrm{C0}}.mem.lm[\![1]\!].mfr.st = st_{\mathrm{fun}}(\textit{dispatcher-proc}) \\
\wedge & \| \, gvar_{\mathrm{lm}}(1, \texttt{dispatcher\_eca}) \, \| = nat(1) \\
\wedge & \| \, gvar_{\mathrm{lm}}(1, \texttt{dispatcher\_edata}) \, \| = nat(0) \\
\wedge & \| \, gvar_{\mathrm{lm}}(1, \texttt{dispatcher\_edpc}) \, \| = nat(0) \\
\wedge & c_{\mathrm{C0}}.prog = \texttt{prog}^4 \\
\longrightarrow & c_{\mathrm{C0}} \in C^4_{\mathrm{C0}}.
\end{array}
$$

The constant $\texttt{prog}^4$ above is a formally defined body of the function `dispatcher`() starting from line 33 of Listing 9.3.

The C0 postcondition of the discussed code portion, defined as a set of C0 configurations C^5_{C0}, differs from its precondition only in the state of the program rest

$$
\begin{array}{ll}
 & c_{\mathrm{C0}} \in C0''\surd(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\
\wedge & structure^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge & structure^{\mathrm{ck}}_{\mathrm{HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge & |c_{\mathrm{C0}}.mem.lm| = 2 \\
\wedge & c_{\mathrm{C0}}.mem.lm[\![0]\!].mfr.st = st_{\mathrm{fun}}(\textit{init-proc}) \\
\wedge & c_{\mathrm{C0}}.mem.lm[\![1]\!].mfr.st = st_{\mathrm{fun}}(\textit{dispatcher-proc}) \\
\wedge & \| \, gvar_{\mathrm{lm}}(1, \texttt{dispatcher\_eca}) \, \| = nat(1) \\
\wedge & \| \, gvar_{\mathrm{lm}}(1, \texttt{dispatcher\_edata}) \, \| = nat(0) \\
\wedge & \| \, gvar_{\mathrm{lm}}(1, \texttt{dispatcher\_edpc}) \, \| = nat(0) \\
\wedge & c_{\mathrm{C0}}.prog = \texttt{prog}^5 \\
\longrightarrow & c_{\mathrm{C0}} \in C^5_{\mathrm{C0}},
\end{array}
$$

where the constant $\texttt{prog}^5$ is a formally defined body of the function `dispatcher`() starting from line 35 of Listing 9.3.

As for the ISA level, the pre- and postconditions coincide. They are defined as sets of ISA configurations C^4_{ISA} and C^5_{ISA}, respectively, and are conjunctions of the following facts: (i) validity of the ISA configuration, (ii) requirement for the system mode to be on, (iii) the ISA code invariants, and (iv) the zero-filled page condition:

$$\begin{array}{ll} & \mathit{isa}\surd(c_{\text{ISA}}) \\ \wedge & \mathit{is\text{-}sys\text{-}exec}_{\text{ISA}}(c_{\text{ISA}}) \\ \wedge & \mathit{code\text{-}inv}(c_{\text{ISA}}) \\ \wedge & \mathit{zfp\text{-}cond}(c_{\text{ISA}}) \\ \longrightarrow & c_{\text{ISA}} \in C^4_{\text{ISA}} \ \wedge \ c_{\text{ISA}} \in C^5_{\text{ISA}}. \end{array}$$

Correctness of this code portion is stated in the following lemma.

Lemma 9.9 (C0 statements of `dispatcher`()) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) an execution sequence *seq* is well-formed, (iii) the hard disk properties hold, (iv) an ISA processor configuration $c_{\text{ISA+DS}}.\mathit{cpu}$ simulates the C0 configuration of the concrete kernel c_{C0}, (v) the $\mathcal{B}$ relation holds between $c_{\text{ISA+DS}}.\mathit{cpu}$ and a configuration of user processes *ups*, and (vi) C0 and ISA preconditions are satisfied, then there exists a number of steps T whose execution brings ISA with deices into a configuration $c'_{\text{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) devices non-interference holds, (ii) the hard disk properties are preserved, (iii) the C0 configuration c'_{C0} is simulated by the ISA configuration $c'_{\text{ISA+DS}}.\mathit{cpu}$, (iv) the $\mathcal{B}$ relation is preserved, (v) value of the variable `SR` remains unchanged, and (vi) the postconditions on the ISA and C0 levels hold:

$$\begin{array}{lll} & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\text{AK}}) & \\ \wedge & \mathit{seq}\surd(\mathit{seq}, c_{\text{ISA+DS}}.\mathit{devs}) & \\ \wedge & \mathit{is\text{-}HD}\surd(c_{\text{ISA+DS}}.\mathit{devs}) & \\ \wedge & \mathit{C0\text{-}sim\text{-}isa}(\mathit{te}_{\text{CK}}(\pi_{\text{AK}}), \mathit{ft}_{\text{CK}}(\pi_{\text{AK}}), c_{\text{C0}}, c_{\text{ISA+DS}}.\mathit{cpu}) & \\ \wedge & \mathcal{B}(\mathit{ups}, c_{\text{ISA+DS}}) & \\ \wedge & c_{\text{ISA+DS}}.\mathit{cpu} \in C^4_{\text{ISA}} & \\ \wedge & c_{\text{C0}} \in C^4_{\text{C0}} & \\ \longrightarrow & \exists\, T, c'_{\text{ISA+DS}}, c'_{\text{C0}} : & \\ & \quad \delta^T_{\text{ISA+DS}}(c_{\text{ISA+DS}}, \mathit{seq}) = c'_{\text{ISA+DS}} & \\ & \wedge \quad \mathit{non\text{-}interf\text{-}dev}(c_{\text{ISA+DS}}.\mathit{devs}, c'_{\text{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) & \\ & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\text{ISA+DS}}.\mathit{devs}) & \\ & \wedge \quad \mathit{C0\text{-}sim\text{-}isa}(\mathit{te}_{\text{CK}}(\pi_{\text{AK}}), \mathit{ft}_{\text{CK}}(\pi_{\text{AK}}), c'_{\text{C0}}, c'_{\text{ISA+DS}}.\mathit{cpu}) & \\ & \wedge \quad \mathcal{B}(\mathit{ups}, c'_{\text{ISA+DS}}) & \\ & \wedge \quad \mathit{val}_{\mathit{sr}}(c'_{\text{ISA+DS}}) = \mathit{val}_{\mathit{sr}}(c_{\text{ISA+DS}}) & \\ & \wedge \quad c'_{\text{ISA+DS}}.\mathit{cpu} \in C^5_{\text{ISA}} & \\ & \wedge \quad c'_{\text{C0}} \in C^5_{\text{C0}}. & \end{array}$$

Isabelle: `cvm/Reset/dispatcher_reset.dispatcher_reset_isa_correct`

Overall Correctness of the Case *Reset*

Putting together correctness statements of five individual parts together we obtain an overall correctness claim of the kernel initialization in case of reset. In the formulation of the overall lemma we will use the implementation invariant of the concrete kernel $\mathit{impl\text{-}inv}_{\text{kernel}}$ (cf. Section 7.1.6) which contains all properties of the ISA postcondition c^5_{ISA} and a part of the properties from the C0 postcondition c^5_{C0}. We define a final C0 postcondition $\mathit{resetPOST}(c_{\text{C0}})$ for the case *reset* which comprises those properties which are not covered by $\mathit{impl\text{-}inv}_{\text{kernel}}$:

$$
\begin{array}{lcl}
\mathit{resetPOST}(c_{\text{C0}}) & \stackrel{\text{def}}{=} & \mathit{left\text{-}stmt}(c_{\text{C0}}.\mathit{prog}) = \mathit{sCall}(\texttt{cup}, \texttt{dispatcher\_kernel}, \\
& & \qquad [\mathit{var}(\texttt{dispatcher\_eca}), \mathit{var}(\texttt{dispatcher\_edata})]) \\
& \wedge & \| \mathit{gvar}_{\text{lm}}(1, \texttt{dispatcher\_eca}) \| = \mathit{nat}(1) \\
& \wedge & \| \mathit{gvar}_{\text{lm}}(1, \texttt{dispatcher\_edata}) \| = \mathit{nat}(0) \\
& \wedge & |c_{\text{C0}}.\mathit{mem.lm}| = 2.
\end{array}
$$

The overall correctness theorem of the case is stated below.

Theorem 9.10 (Correctness of the case *reset*) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed, (iii) the hard disk validity properties hold, and (iv) the processor configuration $c_{\text{ISA+DS}}.\mathit{cpu}$ of the ISA with device is in initial state, then there exists a number of steps T whose execution brings ISA with deices into a configuration $c'_{\text{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk properties are preserved, (iii) the implementation invariant of the concrete kernel holds, (iv) the $\mathcal{B}$ relation holds, (v) the value of the variable `SR` is 8254, (vi) all global variables of the abstract kernel store the corresponding initial values, and (vii) the C0 postcondition for the case *reset* hold:

$$
\begin{array}{lcl}
 & & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\text{AK}}) \\
\wedge & & \mathit{seq}\surd(\mathit{seq}, c_{\text{ISA+DS}}.\mathit{devs}) \\
\wedge & & \mathit{is\text{-}HD}\surd(c_{\text{ISA+DS}}.\mathit{devs}) \\
\wedge & & \mathit{is\text{-}init\text{-}isa}(\pi_{\text{AK}}, c_{\text{ISA+DS}}.\mathit{cpu}) \\
\longrightarrow & & \exists\, T, c'_{\text{ISA+DS}}, c'_{\text{C0}} : \\
 & & \quad \delta^T_{\text{ISA+DS}}(c_{\text{ISA+DS}}, \mathit{seq}) = c'_{\text{ISA+DS}} \\
 & \wedge & \quad \mathit{non\text{-}interf\text{-}dev}'(c_{\text{ISA+DS}}.\mathit{devs}, c'_{\text{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge & \quad \mathit{is\text{-}HD}\surd(c'_{\text{ISA+DS}}.\mathit{devs}) \\
 & \wedge & \quad \mathit{impl\text{-}inv}_{\text{kernel}}(\pi_{\text{AK}}, c'_{\text{C0}}, c'_{\text{ISA+DS}}.\mathit{cpu}) \\
 & \wedge & \quad \mathcal{B}(\lambda \mathit{pid} : \mathit{ups}_{\text{init}}(\mathit{pid}), c'_{\text{ISA+DS}}) \\
 & \wedge & \quad \mathit{val}_{\mathit{sr}}(c'_{\text{ISA+DS}}) = 8254 \\
 & \wedge & \quad \forall s \in \pi_{\text{AK}}.\mathit{gst} : \\
 & & \qquad \mathit{reval}(\mathit{te}_{\text{CK}}(\pi_{\text{AK}}), c'_{\text{C0}}.\mathit{mem}, \mathit{gvar}_{\text{gm}}(s.\mathit{vn})) = \mathit{init}_{\text{val}}(s.\mathit{ty}) \\
 & \wedge & \quad \mathit{resetPOST}(c'_{\text{C0}}).
\end{array}
$$

Isabelle: `cvm/Reset/reset_correct.reset_correct`

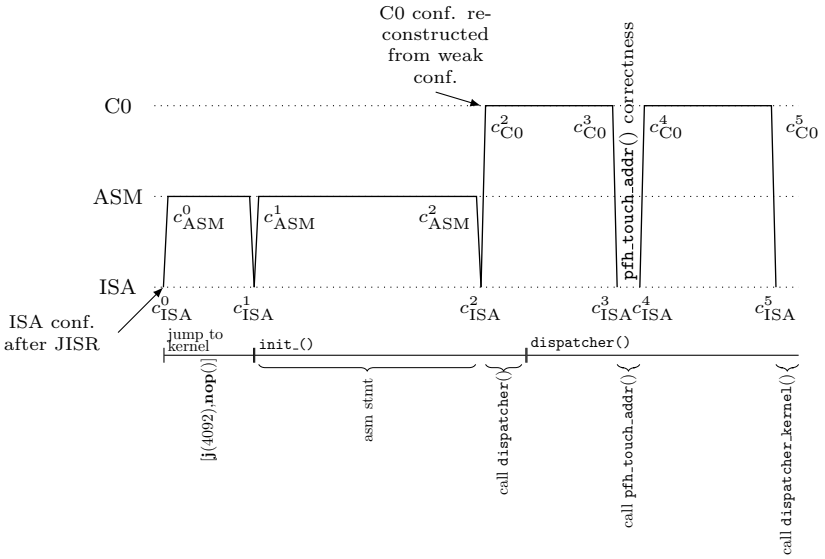

Figure 9.3: Verification diagram of the case *save*.

9.2.3 Correctness of the Case *Save*

The kernel start in the case *save* corresponds to a kernel invocation due to an interrupt occurred during a user run. The context of the interrupted user process has to be saved in kernel data structures. We can also start the kernel in case of a device interrupt during the kernel waiting for interrupts case. In this case the register saving is superfluous, but the kernel is not designed to distinguish these two cases. The following parts of the function `init_`() belong to the case *save*: initialization or restore of global, local, and heap pointers, saving of register values into the process control blocks, copying values of registers needed for interrupt handling into local variables, and the call of the function `dispatcher`(). As for the latter, the following parts of `dispatcher`() are covered by the case: a call of the page-fault handler function `pfh_touch_addr`() which returns the constant `MM_INVALID_ADDRESS` as a result and the further code until the call to `dispatcher_kernel`(). As the result of the page-fault handler indicates, we consider the case that no page faults occur due to an invalid or write-protected access, but rather some other interrupts including a page table length exception. We consider the case when page faults occur later in Section 9.2.5 after we discuss correctness of the case *restore*.

Figure 9.3 sketches the verification process of process-context saving. The main differences from the case *reset* are as follows:

- we do not explicitly create any assembly or C0 configurations, but rather reconstruct them from the weak relations (*C0-sim-isa*$_{\text{weak}}$, *structure*$^{\text{ck}}_{\text{GM}}$, and *structure*$^{\text{ck}}_{\text{HM}}$) which hold before the interrupt occurs, and

- we do not get the relation $\mathcal{B}$ for free exploiting the correctness of the page-fault handler initialization code, but rather prove it ourselves. Note that the $\mathcal{B}$ relation does not hold for the ISA configuration c^0_{ISA}.

Moreover, not only does the $\mathcal{B}$ relation not hold for the configuration c^0_{ISA}, but also neither user implementation invariants $\mathit{impl\text{-}inv}_{\mathrm{user}}$ nor kernel implementation invariants $\mathit{impl\text{-}inv}_{\mathrm{kernel}}$. This is because the ISA execution mode is already changed to system mode, however, the program counters point outside the kernel code, global, local, and heap pointers are not appropriately assigned, and the local stack and the program rest are invalid. In order to resolve this complication we define below a *mixed implementation invariant* which collects all properties that hold at this point. The properties are: (i) the ISA code invariant, (ii) the zero-filled page condition, (iii) a requirement for the ISA machine to run in system mode, (iv) the weak validity of the C0 configuration, (v) the invariants over the concrete kernel global and heap memory structure, (vi) the page-fault handler invariant, and (vii) the weak C0-ISA simulation relation:

$$\begin{aligned}
\mathit{is\text{-}impl\text{-}inv}_{\mathrm{mix}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=}\ & \mathit{isa}\surd(c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{code\text{-}inv}(\pi_{\mathrm{AK}}, c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{zfp\text{-}cond}(c_{\mathrm{ISA}}) \\
\wedge\ & \mathit{is\text{-}sys\text{-}exec}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) \\
\wedge\ & c_{\mathrm{C0}} \in \mathit{C0}_{\mathrm{weak}}\surd(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}})) \\
\wedge\ & \mathit{structure}^{\mathrm{ck}}_{\mathrm{GM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}) \\
\wedge\ & \mathit{structure}^{\mathrm{ck}}_{\mathrm{HM}}(c_{\mathrm{C0}}) \\
\wedge\ & \exists\, c_{\mathrm{PFH}} :\ \mathit{pfh\text{-}inv}(c_{\mathrm{PFH}}, \mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}.\mathit{mem}) \\
\wedge\ & \mathit{C0\text{-}sim\text{-}isa}_{\mathrm{weak}}(\mathit{te}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c_{\mathrm{C0}}, c_{\mathrm{ISA}}).
\end{aligned}$$

As mentioned above, since the mode register of the ISA machine is set to system mode, the $\mathcal{B}$ relation does not hold at this point (cf. the statement of Theorem 9.11). In order to regain it we have to artificially undo the effect of the jump to the interrupt service routine. For this we define the function

$$\mathit{JISR}^{-1} :: C_{\mathrm{ISA+DS}} \mapsto C_{\mathrm{ISA+DS}},$$

$$\mathit{JISR}^{-1}(c_{\mathrm{ISA+DS}}) \stackrel{\mathrm{def}}{=} c'_{\mathrm{ISA+DS}},$$

which returns an updated configuration of the ISA combined system $c'_{\mathrm{ISA+DS}}$ such that:

$$\begin{aligned}
c'_{\mathrm{ISA+DS}}.\mathit{devs} &= c_{\mathrm{ISA+DS}}.\mathit{devs}, \\
c'_{\mathrm{ISA+DS}}.\mathit{cpu.m} &= c_{\mathrm{ISA+DS}}.\mathit{cpu.m}, \\
c'_{\mathrm{ISA+DS}}.\mathit{cpu.gpr} &= c_{\mathrm{ISA+DS}}.\mathit{cpu.gpr}, \\
c'_{\mathrm{ISA+DS}}.\mathit{cpu.dpc} &= c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(\mathit{edpc}), \\
c'_{\mathrm{ISA+DS}}.\mathit{cpu.pc} &= c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(\mathit{epc}), \\
c'_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(r) &= \begin{cases} c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(\mathit{emode}) & \text{if } r = \mathit{mode} \\ c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(\mathit{esr}) & \text{if } r = \mathit{sr} \\ c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(r) & \text{otherwise} \end{cases}.
\end{aligned}$$

The precondition for the case *save* is defined on the ISA level. It is denoted as a set of ISA configurations C^0_{ISA} and is a conjunction of the following facts: (i) the program counters point to the very first address, (ii) if ISA has run in system mode the current process identifier variable corresponds to the kernel (we were in the wait state), otherwise the current process identifier denotes some user and the user implementation

invariant holds, (iii) some interrupt different from the reset signal has occurred and the corresponding bit of the register *eca* is raised, (iv) whenever bits 3 or 4 of the exceptional cause register are raised, which corresponds to a page fault on fetch or load/store, respectively, these signals were caused by a page table length exception. Formally:

$$
\begin{array}{ll}
 & \langle c_{\mathrm{ISA}}.dpc\rangle = 0 \\
\wedge & \langle c_{\mathrm{ISA}}.pc\rangle = 4 \\
\wedge & \langle c_{\mathrm{ISA}}.spr(emode)\rangle = 0 \\
 & \longrightarrow \quad nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}) = 0 \\
\wedge & \langle c_{\mathrm{ISA}}.spr(emode)\rangle \neq 0 \\
 & \longrightarrow \quad \langle c_{\mathrm{ISA}}.spr(emode)\rangle = 1 \\
 & \qquad \wedge \quad 0 < nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}) < 128 \\
 & \qquad \wedge \quad \textit{user-inv}(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup})) \\
\wedge & c_{\mathrm{ISA}}.spr(eca)[0] = 0 \ \wedge \ \exists\, 0 < i < 32 : \ c_{\mathrm{ISA}}.spr(eca)[i] = 1 \\
\wedge & c_{\mathrm{ISA}}.spr(eca)[3] = 1 \ \longrightarrow \ \textit{ptl-excp}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, c_{\mathrm{ISA}}.spr(edpc)) \\
\wedge & c_{\mathrm{ISA}}.spr(eca)[4] = 1 \ \longrightarrow \ \textit{ptl-excp}_{\mathrm{ISA}}(c_{\mathrm{ISA}}, c_{\mathrm{ISA}}.spr(edata)) \\
\longrightarrow & c_{\mathrm{ISA}} \in C^0_{\mathrm{ISA}}
\end{array}
$$

We do not discuss intermediate configurations depicted in Figure 9.3, but present below a final postcondition of process-context saving. The postcondition is defined on the C0 level as a set of C0 configurations $C^5_{\mathrm{C0}}(c_{\mathrm{ISA}})$ which is parametrized by the ISA state from the precondition. The terms which constitute the postcondition are (i) there are two local memory frames, (ii) the values of the variables which are supposed to store values of registers *eca*, *edata*, and *edpc* indeed store these values of the ISA machine c_{ISA}, and (iii) the program rest of the C0 machine is appropriately defined:

$$
\begin{array}{ll}
 & |c_{\mathrm{C0}}.mem.lm| = 2 \\
\wedge & \Vert\, \textit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_eca})\, \Vert = \textit{nat}(\langle c_{\mathrm{ISA}}.spr(eca)\rangle) \\
\wedge & \Vert\, \textit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_edata})\, \Vert = \textit{nat}(\langle c_{\mathrm{ISA}}.spr(edata)\rangle) \\
\wedge & \Vert\, \textit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_edpc})\, \Vert = \textit{nat}(\langle c_{\mathrm{ISA}}.spr(edpc)\rangle) \\
\wedge & c_{\mathrm{C0}}.prog = \texttt{prog}^5 \\
\longrightarrow & c_{\mathrm{C0}} \in C^5_{\mathrm{C0}}(c_{\mathrm{ISA}}),
\end{array}
$$

where $\texttt{prog}^5$ is a formally defined body of the function `dispatcher`() starting from line 35 of Listing 9.3.

Now, we state the correctness theorem for process-context saving.

Theorem 9.11 (Correctness of the case *save*) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed (iii) the hard disk validity properties hold, (iv) the mixed implementation invariant hold, (v) there is sufficient amount of heap memory, (vi) the $\mathcal{B}$ relation hold between the user processes configuration *ups* and and the ISA configuration $c_{\mathrm{ISA+DS}}$ on which the JISR effect is undone, and (vii) the ISA configuration satisfies the preconditions to the case *save*, then there exists a number of steps T whose execution brings ISA with deices into a configuration $c'_{\mathrm{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk properties are preserved,

(iii) the implementation invariant of the concrete kernel holds, (iv) the $\mathcal{B}$ relation holds, (v) the value of the variables `SR` and `cup` remain unchanged, (vi) all global and heap variables of the abstract kernel remain unchanged, and (vii) the C0 postcondition for the case *save* hold:

$$
\begin{array}{ll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \\
\wedge & \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & \mathit{is\text{-}impl\text{-}inv}_{\mathrm{mix}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
\wedge & \mathit{avail}_{\mathrm{heap}}(c_{\mathrm{C0}}) \\
\wedge & \mathcal{B}(\mathit{ups}, \mathit{JISR}^{-1}(c_{\mathrm{ISA+DS}})) \\
\wedge & c_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^0_{\mathrm{ISA}} \\
\longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}} : \\
 & \quad \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev}'(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & \wedge \quad \mathit{impl\text{-}inv}_{\mathrm{kernel}}(\pi_{\mathrm{AK}}, c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
 & \wedge \quad \mathcal{B}(\mathit{ups}, c'_{\mathrm{ISA+DS}}) \\
 & \wedge \quad \mathit{val}_{\mathit{sr}}(c'_{\mathrm{ISA+DS}}) = \mathit{val}_{\mathit{sr}}(c_{\mathrm{ISA+DS}}) \\
 & \wedge \quad \mathit{val}_{\mathit{cup}}(c'_{\mathrm{ISA+DS}}) = \mathit{val}_{\mathit{cup}}(c_{\mathrm{ISA+DS}}) \\
 & \wedge \quad \mathit{abs\text{-}kern\text{-}unch}_{\mathrm{GM,HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) \\
 & \wedge \quad c'_{\mathrm{C0}} \in C^5_{\mathrm{C0}}(c_{\mathrm{ISA+DS}}.\mathit{cpu}).
\end{array}
$$

Isabelle: `cvm/NonReset/nonreset_correct.begin_asm_init_dispatcher_nonreset_isa_correct`

9.2.4 Correctness of the Case *Restore*

Verification of restoring a process-context starts at the point where the abstract kernel ends its execution – in the function `dispatcher`() right after the call to the dispatcher of the abstract kernel. Figure 9.4 sketches the verification process.

The precondition to the case is stated on the C0 level as a set of C0 configurations C^0_{C0} and is rather simple. We require the size of the local stack to be equal to 2 and the program rest to be equal to the constant $\mathtt{prog}^0$, a formally defined body of the function `dispatcher`() starting from line 38 of Listing 9.3:

$$
\begin{array}{ll}
 & |c_{\mathrm{C0}}.\mathit{mem.lm}| = 2 \\
\wedge & c_{\mathrm{C0}}.\mathit{prog} = \mathtt{prog}^0 \\
\longrightarrow & c_{\mathrm{C0}} \in C^0_{\mathrm{C0}}.
\end{array}
$$

We do not define intermediate configuration depicted in Figure 9.3, but rather state the overall correctness theorem for the case. Note, that we do not have special predicates on the C0 level for the overall postcondition because the kernel execution ends.

Theorem 9.12 (Correctness of case *restore*) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed (iii) the hard

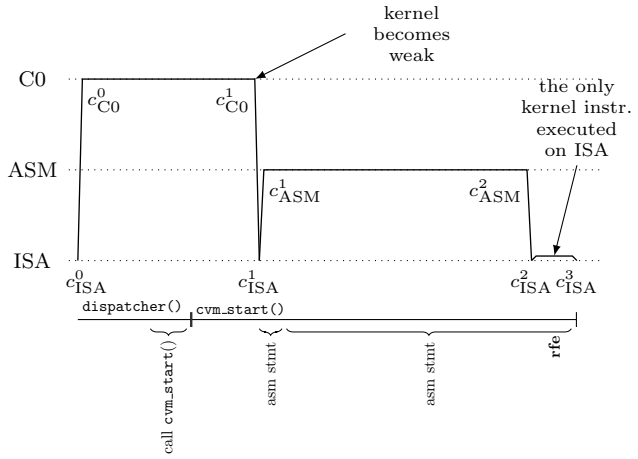

Figure 9.4: Verification diagram of the case *restore*.

disk validity properties hold, (iv) the concrete kernel implementation invariant hold, (v) there is sufficient amount of heap memory, (vi) the $\mathcal{B}$ relation holds, and (vii) the C0 configuration satisfies the preconditions to the case *restore*, then there exists a number of steps T whose execution brings ISA with devices into a configuration $c'_{\text{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) devices non-interference holds, (ii) the hard disk properties are preserved, (iii) the user implementation invariant holds, (iv) the $\mathcal{B}$ relation holds, (v) the value of the variables `SR` and `cup` remain unchanged, and (vi) all global and heap variables of the abstract kernel remain unchanged:

$$
\begin{array}{ll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\text{AK}}) \\
\wedge & \mathit{seq}\surd(\mathit{seq}, c_{\text{ISA+DS}}.\mathit{devs}) \\
\wedge & \mathit{is\text{-}HD}\surd(c_{\text{ISA+DS}}.\mathit{devs}) \\
\wedge & \mathit{impl\text{-}inv}_{\text{kernel}}(\pi_{\text{AK}}, c_{\text{C0}}, c_{\text{ISA+DS}}.\mathit{cpu}) \\
\wedge & \mathit{avail}_{\text{heap}}(c_{\text{C0}}) \\
\wedge & \mathcal{B}(\mathit{ups}, c_{\text{ISA+DS}}) \\
\wedge & c_{\text{C0}} \in C^0_{\text{C0}} \\
\longrightarrow & \exists\, T, c'_{\text{ISA+DS}}, c'_{\text{C0}} : \\
 & \quad \delta^T_{\text{ISA+DS}}(c_{\text{ISA+DS}}, \mathit{seq}) = c'_{\text{ISA+DS}} \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev}(c_{\text{ISA+DS}}.\mathit{devs}, c'_{\text{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\text{ISA+DS}}.\mathit{devs}) \\
 & \wedge \quad \mathit{impl\text{-}inv}_{\text{user}}(\pi_{\text{AK}}, c'_{\text{C0}}, c'_{\text{ISA+DS}}.\mathit{cpu}, \mathit{val}_{sr}(c_{\text{ISA+DS}})) \\
 & \wedge \quad \mathcal{B}(\mathit{ups}, c'_{\text{ISA+DS}}) \\
 & \wedge \quad \mathit{val}_{sr}(c'_{\text{ISA+DS}}) = \mathit{val}_{sr}(c_{\text{ISA+DS}}) \\
 & \wedge \quad \mathit{val}_{cup}(c'_{\text{ISA+DS}}) = \mathit{val}_{cup}(c_{\text{ISA+DS}}) \\
 & \wedge \quad \mathit{abs\text{-}kern\text{-}unch}_{\text{GM,HM}}(\pi_{\text{AK}}, c_{\text{C0}}, c'_{\text{C0}}).
\end{array}
$$

Isabelle: `cvm/Restore/restore_correct.Restore_correct`

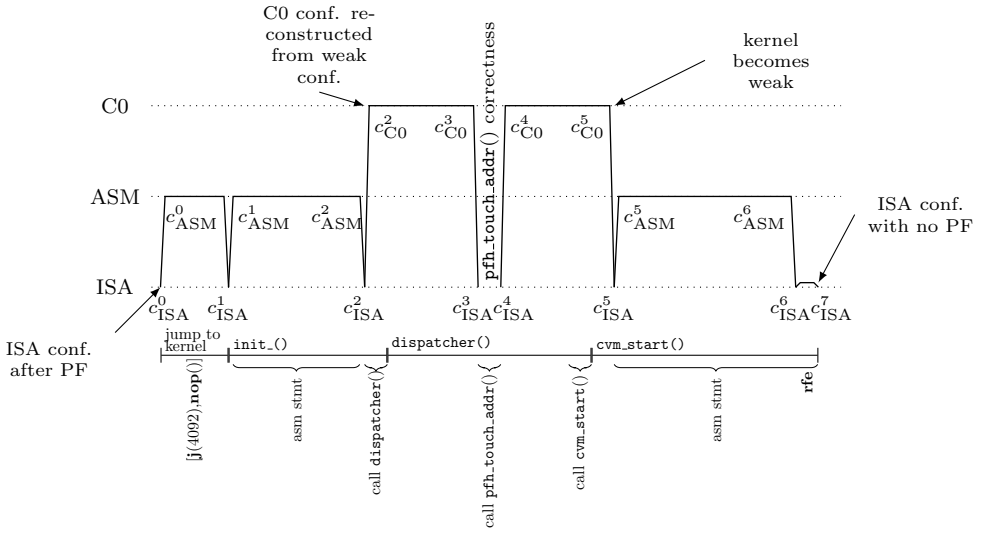

Figure 9.5: Verification diagram of page fault handling.

As follows from the verification diagram the last part on the *restore* case consisting of a single `rfe` instruction is proven at the ISA level, because only at this level the semantics of `rfe` is defined.

9.2.5 Correctness of Page-Fault Handling

The case considered below is a special case of a CVM-based kernel execution without an invocation of the abstract kernel. Consider Figure 9.5 for a sketch of the verification process for this case. For verification of the function `init_`() and the function `cvm_start`() we re-use proofs of the previous cases defined in this chapter so far. Note that, in the CVM dispatcher we clear the variable `dispatcher_eca` just after each call the the handler `pfh_touch_addr`(). By this we skip all further code and continue directly with restoring the interrupted user process via a call to `cvm_start`(). As we will see later it is not enough to show that the configuration c^7_{ISA} is free of the page-fault interrupt that has occurred in the state c^0_{ISA}. In order to guarantee liveness of the system it is necessary to argue that during the next call to the page-fault handler the page that was swapped in this time will be be not swapped out.

The page-fault handler implementation respects the mentioned property because of its page-replacement strategy and the fact that there are more than two user physical pages in the system. On the side of specification the handler algorithm is defined over its configuration c_{PFH}. Suppose we want to claim that the page corresponding to a process *pid* and a virtual address *va* will still be in the physical memory after handling of the next page fault. Let us have a look at the page-fault handler algorithm. If the page fault has occurred because the needed page was not in the physical memory, the handler has to load it from the swap memory. In case there is not a single vacant page in the physical memory, which is indicated by an empty free list, some page has to be evicted.

According to our page-replacement strategy a page at the beginning of the active list has to be swapped out. The formula below claims that in the worst case the page associated with the pair (pid, va) will not be evicted, i.e., it is not the first page in the active list. Hence, it will not be swapped out during the next call to the handler:

$$c_{\mathrm{PFH}}.free = [] \longrightarrow \begin{pmatrix} c_{\mathrm{PFH}}.active[0].pid \\ c_{\mathrm{PFH}}.active[0].vpx \end{pmatrix} \neq \begin{pmatrix} pid \\ px(va) \end{pmatrix}.$$

Now, we need to give this property a meaning on the ISA level. We apply consecutively the simulation relation between the page-fault handler configuration and the C0 configuration, the C0 configuration and the assembly configuration, and finally the assembly configuration and the ISA configuration.

Definition 9.13 (Not a page for eviction) In terms of ISA semantics we define the predicate

$$not\text{-}next\text{-}victim :: C_{\mathrm{ISA}} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B},$$

which holds if the page containing the address va for the process pid will not be swapped out during the next call to the page-fault handler:

$$\begin{aligned} &not\text{-}next\text{-}victim(c_{\mathrm{ISA}}, pid, va) \stackrel{\mathrm{def}}{=} \\ &\quad nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{freelist}) = 0 \longrightarrow \\ &\qquad \begin{pmatrix} nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{activelist}) + 0) \\ nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{activelist}) + 4) \end{pmatrix} \neq \begin{pmatrix} pid \\ px(va) \end{pmatrix}. \end{aligned}$$

Isabelle: `cvm/additional/pfh_lemmas.not_next_victim`

Now we can formulate the specification for the case of page-fault handling. Before that, let us introduce one more predicate. It is a page-fault predicate defined on the software level:

$$\begin{aligned} not\text{-}valid\text{-}or\text{-}protected(c_{\mathrm{ISA}}, pid, va) \quad &\stackrel{\mathrm{def}}{=} \quad p(get\text{-}pte(c_{\mathrm{ISA}}, pid, px(va))) = 1 \\ &\vee \quad v(get\text{-}pte(c_{\mathrm{ISA}}, pid, px(va))) = 0. \end{aligned}$$

It has the meaning that a page-fault takes place if a page corresponding to a process pid and a virtual address va is either protected or does not reside in the main memory.

The precondition to the considered case is formulated on the ISA level as a set of ISA configurations C^0_{ISA}. The predicate below collect the following facts: (i) the program counters point to the very first address, (ii) the exception mode is user, (iii) the current process identifier corresponds to some user process, (iv) the user invariant holds, (v) a page fault either on fetch or on load/store takes place, and (vi) for both kinds of page

fault no page table length exception occurs. Formally:

$$
\begin{array}{ll}
 & \langle c_{\mathrm{ISA}}.dpc\rangle = 0 \\
\wedge & \langle c_{\mathrm{ISA}}.pc\rangle = 4 \\
\wedge & \langle c_{\mathrm{ISA}}.spr(emode)\rangle = 1 \\
\wedge & 0 < nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}) < 128 \\
\wedge & user\text{-}inv(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup})) \\
\wedge & \forall\, i \leq 2 :\ c_{\mathrm{ISA}}.spr(eca)[i] = 0 \ \wedge\ \exists\, i \in \{3,4\} :\ c_{\mathrm{ISA}}.spr(eca)[i] = 1 \\
\wedge & c_{\mathrm{ISA}}.spr(eca)[3] = 1 \\
 & \quad \longrightarrow \ \neg ptl\text{-}excp_{\mathrm{ISA}}(c_{\mathrm{ISA}}, c_{\mathrm{ISA}}.spr(edpc)) \\
 & \qquad \wedge \ \ not\text{-}valid\text{-}or\text{-}protected(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), \langle c_{\mathrm{ISA}}.spr(edpc)\rangle) \\
\wedge & c_{\mathrm{ISA}}.spr(eca)[4] = 1 \\
 & \quad \longrightarrow \ \neg ptl\text{-}excp_{\mathrm{ISA}}(c_{\mathrm{ISA}}, c_{\mathrm{ISA}}.spr(edata)) \\
 & \qquad \wedge \ \ not\text{-}valid\text{-}or\text{-}protected(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), \langle c_{\mathrm{ISA}}.spr(edata)\rangle) \\
\longrightarrow & c_{\mathrm{ISA}} \in C^0_{\mathrm{ISA}}.
\end{array}
$$

As we deal with two calls of the page-fault handler we apply its correctness theorem twice. Hence, we need to guarantee that the predicate *not-next-victim* holds not only for the process identifier *pid* and the virtual address *va* at which the first page fault takes place, but actually if that predicate holds for any other address of this process then the corresponding page is not swapped out during the kernel execution. This is expressed in the postcondition below.

The postcondition is formulated on the ISA level as a set of ISA configurations $C^7_{\mathrm{ISA}}(c^0_{\mathrm{ISA}})$. Besides the mentioned property, the postcondition it claims the following: (i) if a page fault on fetch occurred then the page addressed by the counter *edpc* for the current process will not be evicted during the next call to the page-fault handler and this page is loaded to the main memory, and (ii) if a page fault on load/store occurred then the page corresponding to the address stored in the register *edata* and the current process will not be evicted during the next call to the page-fault handler and this page is loaded to the main memory. Formally:

$$
\begin{array}{ll}
 & c^0_{\mathrm{ISA}}.spr(eca)[3] = 1 \\
 & \longrightarrow \ not\text{-}next\text{-}victim(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), \langle c_{\mathrm{ISA}}.spr(edpc)\rangle) \\
 & \quad \wedge \ \ \neg not\text{-}valid\text{-}or\text{-}protected(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), \langle c_{\mathrm{ISA}}.spr(edpc)\rangle) \\
\wedge & c^0_{\mathrm{ISA}}.spr(eca)[4] = 1 \\
 & \longrightarrow \ not\text{-}next\text{-}victim(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), \langle c_{\mathrm{ISA}}.spr(edata)\rangle) \\
 & \quad \wedge \ \ \neg not\text{-}valid\text{-}or\text{-}protected(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), \langle c_{\mathrm{ISA}}.spr(edata)\rangle) \\
\wedge & \forall addr : \\
 & \quad px(addr) \leq \langle c^0_{\mathrm{ISA}}.spr(ptl)[19,0]\rangle \\
 & \wedge \ \ not\text{-}next\text{-}victim(c^0_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), addr) \\
 & \longrightarrow \ \ get\text{-}pte(c_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), px(addr)) \\
 & \quad = \ get\text{-}pte(c^0_{\mathrm{ISA}}, nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{cup}), px(addr)) \\
\longrightarrow & c_{\mathrm{ISA}} \in C^7_{\mathrm{ISA}}(c^0_{\mathrm{ISA}}).
\end{array}
$$

Finally, we state the correctness theorem for the case of page-fault handling.

Theorem 9.14 (Correctness of page-fault handling) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed (iii) the hard disk validity properties hold, (iv) the mixed implementation invariants hold, (v) there is sufficient amount of heap memory, (vi) the $\mathcal{B}$ relation holds between the configuration of user processes *ups* and the ISA configuration $c_{\mathrm{ISA+DS}}$ on which the effect of JISR is undone, and (vii) the ISA configuration satisfies the preconditions to the case of page-fault handling, then there exists a number of steps T whose execution brings ISA with devices into a configuration $c'_{\mathrm{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk properties are preserved, (iii) the user implementation invariant holds, (iv) the $\mathcal{B}$ relation holds, (v) the value of the variables `SR` and `cup` remain unchanged, (vi) all global and heap variables of the abstract kernel remain unchanged, and (vii) the postcondition for the case of page-fault handling holds:

$$
\begin{array}{lll}
 & & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \\
\wedge & & \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & & \mathit{is\text{-}impl\text{-}inv}_{\mathrm{mix}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
\wedge & & \mathit{avail}_{\mathrm{heap}}(c_{\mathrm{C0}}) \\
\wedge & & \mathcal{B}(\mathit{ups}, \mathit{JISR}^{-1}(c_{\mathrm{ISA+DS}})) \\
\wedge & & c_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^0_{\mathrm{ISA}} \\
\longrightarrow & & \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}}, : \\
 & & \quad \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} \\
 & \wedge & \quad \mathit{non\text{-}interf\text{-}dev}'(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge & \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & \wedge & \quad \mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{val}_{sr}(c_{\mathrm{ISA+DS}})) \\
 & \wedge & \quad \mathcal{B}(\mathit{ups}, c'_{\mathrm{ISA+DS}}) \\
 & \wedge & \quad \mathit{val}_{sr}(c'_{\mathrm{ISA+DS}}) = \mathit{val}_{sr}(c_{\mathrm{ISA+DS}}) \\
 & \wedge & \quad \mathit{val}_{cup}(c'_{\mathrm{ISA+DS}}) = \mathit{val}_{cup}(c_{\mathrm{ISA+DS}}) \\
 & \wedge & \quad \mathit{abs\text{-}kern\text{-}unch}_{\mathrm{GM,HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) \\
 & \wedge & \quad c'_{\mathrm{ISA+DS}}.\mathit{cpu} \in C^7_{\mathrm{ISA}}(c_{\mathrm{ISA+DS}}.\mathit{cpu}).
\end{array}
$$

Isabelle: `cvm/UserStep/user_pfh_correct.init_dispatcher_pfhta_dispatcher_start_isa_correct`

9.3 Primitives

This section elaborates on the verification of the CVM primitives. In the frame of this work three primitives were verified: `cvm_copy`(), `cvm_get_vm_word`(), and `cvm_set_vm_word`(). However, the last two primitives were excluded during a recent redesign of the CVM code from the current code release. Partially because of that, and also due to their simplicity — there no are details in their proofs that do not appear in the proof of `cvm_copy`() — we do not discuss them in this thesis. In the remainder of this chapter we first present the implementation of the CVM primitive for copying data between user processes, and then prove its correctness.

Listing 9.5: Primitive cvm_copy().

```
1  int cvm_copy(unsigned int pid1, unsigned int pid2, unsigned int sa1, unsigned int sa2,
       unsigned int amount)
2  {
3    unsigned int pa1;
4    unsigned int pa2;
5    unsigned int chunk_size;
6    while (amount > 0u)
7    {
8      if ((sa1 & (PAGE_SIZE - 1)) < (sa2 & (PAGE_SIZE - 1))) /* compute the next chunk to copy */
9      {                                                      /* -- at max until the next page boundary */
10       chunk_size = PAGE_SIZE - (sa2 & (PAGE_SIZE - 1));
11     }
12     else
13     {
14       chunk_size = PAGE_SIZE - (sa1 & (PAGE_SIZE - 1));
15     }
16     if (chunk_size > amount)     /* ... but not more than requested */
17     {
18       chunk_size = amount;
19     }
20      pa1 = pfh_touch_addr(pid1, sa1, MM_READ, 1u); /* ensure that the source and */
21     if (chunk_size == PAGE_SIZE)                   /* destination page are both in PM */
22     {
23       pa2 = pfh_touch_addr(pid2, sa2, MM_OVERWRITE, 0u);
24     }
25     else
26     {
27       pa2 = pfh_touch_addr(pid2, sa2, MM_WRITE, 0u);
28     }
29     assembler(
30       loadlocal(r11, pa1);
31       loadlocal(r12, pa2);
32       loadlocal(r13, chunk_size);
33       lw(r3, r11, 0);
34       sw(r3, r12, 0);
35       addi(r11, r11, 4);
36       subi(r13, r13, 4);
37       bnez(r13, -20);
38       addi(r12, r12, 4);
39     );
40     amount = amount - chunk_size;
41     sa1    = sa1 + chunk_size;
42     sa2    = sa2 + chunk_size;
43   }
44   return 0;
45 }
```

9.3.1 Implementation of cvm_copy()

The cvm_copy() primitive whose source code is presented in Listing 9.5 is designed to copy amount bytes from a process pid1 at address sa1 to a process pid2 at address sa2.

There are two basic observations which constitute the idea of the implemented algorithm: (i) for each step of the algorithm both pages, from and to which we copy, must be present in the physical memory, and (ii) copying must respect page borders. In order to support these points we declare at lines 3–5 variables pa1 and pa2 which will store translated physical addresses between which we will copy, and a variable chunk_size which is supposed to store the amount of data which could be copied in a single iteration respecting the page borders.

In a loop (lines 6–43) until amount of bytes is processed we compute the size chunk_size of a portion to be copied in the current iteration (lines 8–19). Further, at lines 20–28 we ensure that the source and destination pages reside in the physical

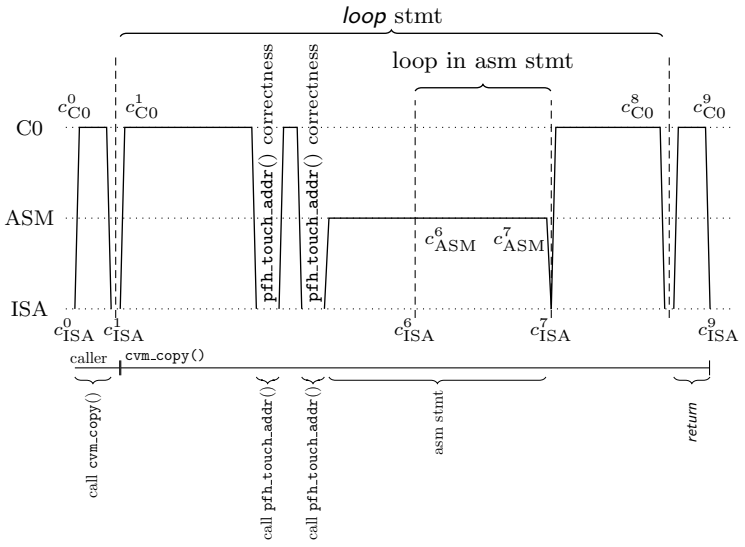

Figure 9.6: Verification diagram of `cvm_copy()` primitive.

memory. This is achieved by calling the page-fault handler. At line 20 we invoke it to obtain the source physical address `pa1` and to guarantee that the page storing `pa1` resides in the main memory. Moreover, the last argument in the handler's call is 1, which indicates that this page will survive one additional call to the page-fault handler. At lines 21-28 we obtain in a similar fashion the translated destination address `pa2`. The case distinction on `chunk_size` takes place to profit from an optimization feature of the page-fault handler: in case the whole page will be rewritten we pass an indicator `MM_OVERWRITE` to the handler. The handler then will not swap in data at this page.

Having `chunk_size` computed and both source and destination translated addresses we proceed with the physical copying of data. For this an assembly statement is used (lines 29–39). The assembly portion implements a simple loop of word-by-word copying. Finally at lines 40–42 we decrease the remaining `amount` to be copied by `chunk_size` while increasing addresses `sa1` and `sa2` by the same number.

9.3.2 Correctness of `cvm_copy()`

Figure 9.6 depicts the verification process of the primitive `cvm_copy()`. Note that, the function contains a nested loop, the inner loop of which is coded in assembly. We start the correctness proof from the inner loop, then combine it with the remaining code of the outer loop's body and proceed with the outer loop. The assembly loop is trivial: while verifying it we make induction on the value of register 13 divided by 4. The outer loop copies portions of data of size `chunk_size`. Here, we distinguish two situations: `chunk_size` is equal to the page size or less than it. In case `chunk_size` $<$ `PAGE_SIZE`, the page-fault handler correctness theorem guarantees that the $\mathcal{B}$-relation is preserved. In case `chunk_size` $=$ `PAGE_SIZE`, the page-fault handler preserves the $\mathcal{B}$-relation for all processes and pages except the current page of the current process. After the execution of the assembly code the desired user process is updated and the $\mathcal{B}$-relation holds.

Let pid_{from}, pid_{to}, a_{from}, a_{to} and n be the values of the variables `pid1`, `pid2`, `sa1`, `sa2`

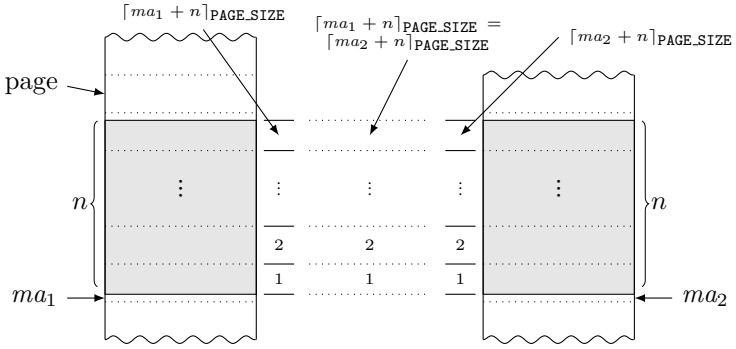

Figure 9.7: Computing the ranking function for `cvm_copy`() loop. Case $ma_1 = ma_2$.

and `chunk_size`, correspondingly, after the assembly loop execution:

$$\begin{aligned}
nat_{\text{vars}}(c^7_{\text{ISA}}, ad_{pid1}) &= pid_{\text{from}},\\
nat_{\text{vars}}(c^7_{\text{ISA}}, ad_{pid2}) &= pid_{\text{to}},\\
nat_{\text{vars}}(c^7_{\text{ISA}}, ad_{sa1}) &= a_{\text{from}},\\
nat_{\text{vars}}(c^7_{\text{ISA}}, ad_{sa2}) &= a_{\text{to}},\\
nat_{\text{vars}}(c^7_{\text{ISA}}, ad_{chunk_size}) &= n.
\end{aligned}$$

Then the updated process configuration is changed only in the memory component for process pid_{to}:

$$ups'(pid_{\text{to}}).m = \textit{mem-part-copy}(ups(pid_{\text{to}}).m, a_{\text{to}}, ups(pid_{\text{from}}).m, a_{\text{from}}, n).$$

Note that the code contains two consecutive calls of the page-fault handler to obtain two physical addresses `pa1` and `pa2` and to load the corresponding physical pages if necessary. In order to show that the first physical page is not swapped out during the second call we use the same approach as for handling the user page fault: we use the predicate *not-next-victim* (cf. Definition 9.13).

Clearly, we need to show that the memory content of the process pid_{to} is modified in an intended way, and for all other processes nothing is changed. For this we prove the fact that any two different pairs of process identifiers and virtual page indices are mapped to different physical addresses in the ISA machine. Here we use Lemma 7.22 in Starostin's thesis [105] which proves the same on the page-fault handler abstraction level.

As for the outer loop, it is quite tricky to choose the right induction variable. The loop ranking function f depends on the value `amount` of data to be copied as well as the start addresses `sa1` and `sa2` (actually, not on the values of addresses but on their alignments). While working with the physical memory in a single loop iteration we can copy only addresses that fit in the same page for both processes. During the next iteration we load the successor page for one or both processes. In general, for an amount of bytes to be copied n and two numbers ma_1 and ma_2 that are start addresses modulo `PAGE_SIZE`:

$$\begin{aligned}
ma_1 &= sa_1 \bmod \texttt{PAGE_SIZE},\\
ma_2 &= sa_2 \bmod \texttt{PAGE_SIZE},
\end{aligned}$$

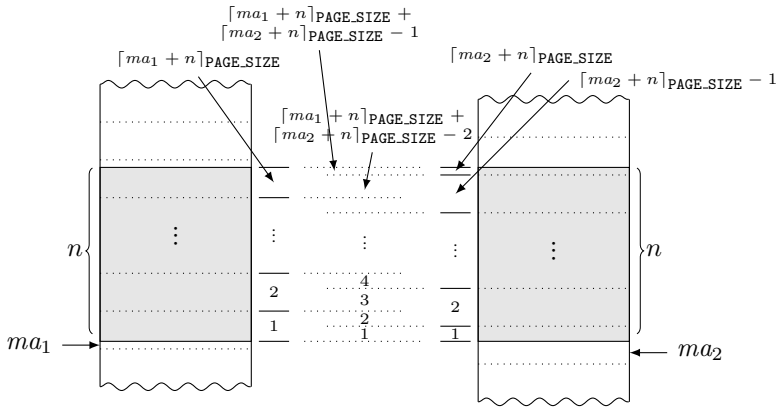

Figure 9.8: Computing the ranking function for `cvm_copy`() loop. Case $ma_1 \neq ma_2$.

the following two situations are possible:

Case 1. The addresses are equal modulo the page size (cf. Figure 9.7): $ma_1 = ma_2$. In this case the loop will be executed as many times as the number of different pages that fit in the copied region:

$$\begin{aligned} f &= \lceil ma_1 + n \rceil_{\texttt{PAGE\_SIZE}} \\ &= \lceil ma_2 + n \rceil_{\texttt{PAGE\_SIZE}}. \end{aligned}$$

Case 2. The addresses modulo page size are not equal (cf. Figure 9.8): $ma_1 \neq ma_2$. As depicted in the figure the number of loop iteration is equal to following:

$$f = \lceil ma_1 + n \rceil_{\texttt{PAGE\_SIZE}} + \lceil ma_2 + n \rceil_{\texttt{PAGE\_SIZE}} - 1.$$

We summarize our argument on the ranking function in the definition below.

Definition 9.15 (Ranking function of `cvm_copy`()) For an amount of bytes to be copied n and two start addresses sa_1 and sa_2 the function

$$measure_{\text{copy}} :: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$$

computes the number of outer loop iterations to be executed in the function `cvm_copy`():

$$measure_{\text{copy}}(n, sa_1, sa_2) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } n = 0 \\ \lceil ma_1 + n \rceil_{\texttt{PAGE\_SIZE}} & \text{if } ma_1 = ma_2 \text{ ,} \\ \lceil ma_1 + n \rceil_{\texttt{PAGE\_SIZE}} + \lceil ma_2 + n \rceil_{\texttt{PAGE\_SIZE}} - 1 & \text{otherwise} \end{cases}$$

where $ma_1 = sa_1 \bmod \texttt{PAGE\_SIZE}$ and $ma_2 = sa_2 \bmod \texttt{PAGE\_SIZE}$.

Isabelle: `cvm/cvm_copy/cvm_copy_c0_loop_ind.cvm_copy_measure`

We use the defined ranking function in the following way. Let n, sa_1, and sa_2 store the following values:

$$\begin{aligned} value_{\mathrm{g}}(mc, \mathit{gvar}_{\mathrm{lm}}(|mc.lm| - 1, \mathtt{amount})) &= \mathit{nat}(n), \\ value_{\mathrm{g}}(mc, \mathit{gvar}_{\mathrm{lm}}(|mc.lm| - 1, \mathtt{sa1})) &= \mathit{nat}(sa_1), \\ value_{\mathrm{g}}(mc, \mathit{gvar}_{\mathrm{lm}}(|mc.lm| - 1, \mathtt{sa2})) &= \mathit{nat}(sa_2). \end{aligned}$$

Then the total number of the outer loop iterations is equal to

$$measure_{\mathrm{copy}}(n, sa_1, sa_2).$$

Now, our goal is to justify correctness of our choice for the ranking function. We will prove that the value of this function decreases with each loop iteration. Recall that during a single iteration the amount to be copied is decremented by `chunk_size` while the start address for copying are incremented by the same value. Formally the size of the chunk is computed as follows:

$$\begin{aligned} \mathit{chunk\_size} = \mathtt{min}(\mathtt{PAGE\_SIZE} - \mathtt{max}(&sa_1 \bmod \mathtt{PAGE\_SIZE}, \\ &sa_2 \bmod \mathtt{PAGE\_SIZE}), \\ n). \end{aligned}$$

The following lemma shows that the chosen ranking function strictly decreases.

Lemma 9.16 (Correctness of ranking function for `cvm_copy()`) Let us denote sa_1 mod `PAGE_SIZE` by ma_1, and sa_2 mod `PAGE_SIZE` by ma_2. Then

$$\begin{aligned} &measure_{\mathrm{copy}}(n - \mathtt{min}(\mathtt{PAGE\_SIZE} - \mathtt{max}(ma_1, ma_2), n), \\ &\qquad sa_1 + \mathtt{min}(\mathtt{PAGE\_SIZE} - \mathtt{max}(ma_1, ma_2), n), \\ &\qquad sa_2 + \mathtt{min}(\mathtt{PAGE\_SIZE} - \mathtt{max}(ma_1, ma_2), n)) \\ &= measure_{\mathrm{copy}}(n, sa_1, sa_2) - 1. \end{aligned}$$

Isabelle: `cvm/cvm_copy/cvm_copy_c0_loop_ind.cvm_copy_measure_mono`

Proof. We make a case distinction on the fact whether a page border is crossed.

Case 1. $n + \mathtt{max}(ma_1, ma_2) \leq \mathtt{PAGE\_SIZE}$.

$$\begin{aligned} &measure_{\mathrm{copy}}(n - \mathtt{min}(\mathtt{PAGE\_SIZE} - \mathtt{max}(ma_1, ma_2), n), \\ &\qquad sa_1 + \mathtt{min}(\mathtt{PAGE\_SIZE} - \mathtt{max}(ma_1, ma_2), n), \\ &\qquad sa_2 + \mathtt{min}(\mathtt{PAGE\_SIZE} - \mathtt{max}(ma_1, ma_2), n)) \\ = \;& measure_{\mathrm{copy}}(n - n, sa_1 + n, sa_2 + n) \\ = \;& 0 \\ = \;& measure_{\mathrm{copy}}(n, sa_1, sa_2) - 1. \end{aligned}$$

Case 2. $n + \mathtt{max}(ma_1, ma_2) > \mathtt{PAGE\_SIZE}$

Case 2.1. $ma_1 = ma_2$

$$\begin{aligned}
& measure_{\text{copy}}(n - \texttt{min}(\texttt{PAGE_SIZE} - \texttt{max}(ma_1, ma_2), n), \\
& \qquad sa_1 + \texttt{min}(\texttt{PAGE_SIZE} - \texttt{max}(ma_1, ma_2), n), \\
& \qquad sa_2 + \texttt{min}(\texttt{PAGE_SIZE} - \texttt{max}(ma_1, ma_2), n)) \\
= \quad & measure_{\text{copy}}(n - (\texttt{PAGE_SIZE} - ma_1), \\
& \qquad sa_1 + (\texttt{PAGE_SIZE} - ma_1), \\
& \qquad sa_2 + (\texttt{PAGE_SIZE} - ma_1)) \\
= \quad & \lceil (sa_1 + (\texttt{PAGE_SIZE} - ma_1)) \bmod \texttt{PAGE_SIZE} \\
& \qquad + n - (\texttt{PAGE_SIZE} - ma_1) \rceil_{\texttt{PAGE_SIZE}} \\
= \quad & \lceil n - (\texttt{PAGE_SIZE} - ma_1) \rceil_{\texttt{PAGE_SIZE}} \\
= \quad & \lceil ma_1 + n \rceil_{\texttt{PAGE_SIZE}} - 1 \\
= \quad & measure_{\text{copy}}(n, sa_1, sa_2) - 1
\end{aligned}$$

Case 2.2. $ma_1 > ma_2$

$$\begin{aligned}
& measure_{\text{copy}}(n - \texttt{min}(\texttt{PAGE_SIZE} - \texttt{max}(ma_1, ma_2), n), \\
& \qquad sa_1 + \texttt{min}(\texttt{PAGE_SIZE} - \texttt{max}(ma_1, ma_2), n), \\
& \qquad sa_2 + \texttt{min}(\texttt{PAGE_SIZE} - \texttt{max}(ma_1, ma_2), n)) \\
= \quad & measure_{\text{copy}}(n - (\texttt{PAGE_SIZE} - ma_1), \\
& \qquad sa_1 + (\texttt{PAGE_SIZE} - ma_1), \\
& \qquad sa_2 + (\texttt{PAGE_SIZE} - ma_1)) \\
= \quad & \lceil (sa_1 + (\texttt{PAGE_SIZE} - ma_1)) \bmod \texttt{PAGE_SIZE} \\
& \qquad + (n - (\texttt{PAGE_SIZE} - ma_1)) \rceil_{\texttt{PAGE_SIZE}} \\
& + \lceil (sa_2 + (\texttt{PAGE_SIZE} - ma_1)) \bmod \texttt{PAGE_SIZE} \\
& \qquad + (n - (\texttt{PAGE_SIZE} - ma_1)) \rceil_{\texttt{PAGE_SIZE}} \\
& - 1 \\
= \quad & \lceil (n - (\texttt{PAGE_SIZE} - ma_1)) \rceil_{\texttt{PAGE_SIZE}} \\
& + \lceil ma_2 + (\texttt{PAGE_SIZE} - ma_1) + (n - (\texttt{PAGE_SIZE} - ma_1)) \rceil_{\texttt{PAGE_SIZE}} \\
& - 1 \\
= \quad & \lceil ma_1 + n \rceil_{\texttt{PAGE_SIZE}} + \lceil ma_2 + n \rceil_{\texttt{PAGE_SIZE}} - 1 - 1 \\
= \quad & measure_{\text{copy}}(n, sa_1, sa_2) - 1
\end{aligned}$$

□

The lemma above is used to conclude that the primitive `cvm_copy()` terminates. It remains to show its functional correctness.

Let us agree on the following notation. For a C0 expression e and a value v we will write

$$\| e \|_{te,mc} = v$$

to denote that the expression is initialized with respect to a type environment te and a memory configuration mc and has the value of v:

$$\begin{aligned}
& \textit{is-initialized}(te, mc, e) \\
\wedge \quad & \textit{reval}(te, mc, e) = \lfloor v \rfloor.
\end{aligned}$$

The precondition to the primitive `cvm_copy`() is formulated as a set of C0 configurations

$$C^0_{\mathrm{C0}}(pid^0_1, pid^0_2, sa^0_1, sa^0_2, amount^0),$$

where parameters denote the initial values of the primitive's parameters. The precondition requires the following statements to be satisfied: (i) there is sufficient amount of heap memory to execute the primitive, (ii) the local memory stack is appropriately bounded, (iii) the statement in the program rest to be executed next is a call to the primitive with a corresponding list of parameters, (iv) there is a variable on the local stack where the result of the call will be stored, (v) the parameters passed to the primitive are correctly typed, and (vi) whenever we call the primitive to copy some positive amount of data the following must be satisfied: we intend to copy between different user processes, the amount to be copied as well as start addresses are word-aligned, and both processes have enough virtual memory to perform the copying. Formally:

$$\begin{array}{lll}
 & avail_{\mathrm{heap}}(c_{\mathrm{C0}}) & \\
\wedge & asize_{\mathrm{st^*}}(sc(c_{\mathrm{C0}}.mem).lst) + 184 \leq \mathtt{ABASE}_{\mathrm{hm}} - \mathtt{ABASE}_{\mathrm{lm}} & \\
\wedge & stmt(c_{\mathrm{C0}}) = \mathit{sCall}(vn, \mathtt{cvm_copy}, params) & \\
\wedge & vn \in vns(lst_{\mathrm{top}}(c_{\mathrm{C0}}.mem)) & \\
\wedge & \forall\, i < 5 :\ \Vert\!\downarrow params[i] \downarrow\!\Vert = \mathit{nat}([pid^0_1, pid^0_2, sa^0_1, sa^0_2, amount^0][i]) & \\
\wedge & amount^0 \neq 0 & \\
 & \quad \longrightarrow \quad pid^0_1 \neq pid^0_2 \ \wedge\ 0 < pid^0_1 < 128 \ \wedge\ 0 < pid^0_2 < 128 & \\
 & \qquad \wedge \quad amount^0 \bmod 4 = 0 \ \wedge\ sa^0_1 \bmod 4 = 0 \ \wedge\ sa^0_2 \bmod 4 = 0 & \\
 & \qquad \wedge \quad sa^0_1 + amount^0 - 1 < (ptl_1 + 1) * \mathtt{PAGE_SIZE} & \\
 & \qquad \wedge \quad sa^0_2 + amount^0 - 1 < (ptl_2 + 1) * \mathtt{PAGE_SIZE} & \\
\longrightarrow & c_{\mathrm{C0}} \in C^0_{\mathrm{C0}}(pid^0_1, pid^0_2, sa^0_1, sa^0_2, amount^0). &
\end{array}$$

Above ptl_1 and ptl_2 correspond to the page table lengths of respective processes. They are define as follows. Let *gvar-ptl*(*pid*) be a C0 variable which stores the page table length value of the process *pid*, i.e., `pcb`[*pid*].`exception_frame`[PTL_{EF}]:

$$\begin{array}{l}
gvar\text{-}ptl(pid) = \\
\qquad \mathit{gvar}_{\mathrm{arr}}(\mathit{gvar}_{\mathrm{str}}(\mathit{gvar}_{\mathrm{arr}}(\mathit{gvar}_{\mathrm{gm}}(\mathtt{pcb}), pid), \mathtt{exception_frame}), PTL_{\mathrm{EF}}).
\end{array}$$

Then ptl_1 and ptl_2 are the values of this variable for process identifiers pid_1 and pid_2:

$$\begin{array}{rcl}
value_{\mathrm{g}}(c_{\mathrm{C0}}.mem, gvar\text{-}ptl(pid_1)) & = & \mathit{int}(ptl_1), \\
value_{\mathrm{g}}(c_{\mathrm{C0}}.mem, gvar\text{-}ptl(pid_2)) & = & \mathit{int}(ptl_2).
\end{array}$$

Restrictions over the local stack follow from the fact that we need to have enough space to call the primitive `cvm_copy`() and its subcalls to `pfh_touch_addr`() as well as subroutines of the page-fault handler. We estimate that for the worst execution scenario the page-fault handler function needs 136 bytes:

$$(ft_{\mathrm{CK}}(\pi_{\mathrm{AK}}), \mathtt{pfh\_touch\_addr}, 136) \in SE.$$

Hence, the estimation for the local memory stack size is as follows:

$$asize_{\mathrm{st}}(st_{\mathrm{fun}}(cvm\text{-}copy\text{-}proc)) + 136 = 184.$$

According to the CVM model effects of primitives are specified within the CVM transition function. So, we have already defined the semantics of the primitive `cvm_copy`() as a function

$$exec_{\texttt{cvm\_copy}}(c_{\mathrm{CVM}}, pid_1, pid_2, sa_1, sa_2, amount)$$

in Section 5.4. Let us denote by

$$copy(ups, pid_1, pid_2, sa_1, sa_2, amount),$$

the part of this function, which takes and returns only the user processes configuration. Thus, for the postcondition it remains only to claim the C0 call of the primitive is correctly processes and the crucial memory invariants hold. The postcondition is formulated as a set of C0 configurations C^9_{C0}:

$$\begin{array}{ll} & \mathit{is\text{-}C0mem\text{-}inv}(te_{\mathrm{CK}}(\pi_{\mathrm{AK}}), c^0_{\mathrm{C0}}.mem, c_{\mathrm{C0}}.mem, [], vn, \mathsf{int}(0)) \\ \wedge & c_{\mathrm{C0}}.cProg = \mathit{rem\text{-}1st\text{-}stmt}(c^0_{\mathrm{C0}}.cProg) \\ \longrightarrow & c_{\mathrm{C0}} \in C^9_{\mathrm{C0}}. \end{array}$$

Finally, we state the correctness theorem of the primitive.

Theorem 9.17 (Correctness of `cvm_copy`()) Assume that (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed (iii) the hard disk validity properties hold, (iv) the kernel implementation invariants hold, (v) the $\mathcal{B}$-relation holds, and (vi) the C0 configuration satisfies the preconditions to the primitive `cvm_copy`(), then there exists a number of steps T whose execution brings ISA with deices into a configuration $c'_{\mathrm{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk properties are preserved, (iii) the kernel implementation invariants are preserved, (iv) the $\mathcal{B}$-relation holds between the user process configuration on which the semantics of the primitive is applied and the updated ISA configuration, (v) the value of the variable `SR` remain unchanged, (vi) all global and heap variables of the abstract kernel remain unchanged, and (vii) the postcondition of the primitive holds:

$$\begin{array}{lll} & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) & \\ \wedge & \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\ \wedge & \mathit{is\text{-}HD}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs}) & \\ \wedge & \mathit{impl\text{-}inv}_{\mathrm{kernel}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c_{\mathrm{ISA+DS}}.cpu) & \\ \wedge & \mathcal{B}(ups, c_{\mathrm{ISA+DS}}) & \\ \wedge & c_{\mathrm{C0}} \in C^0_{\mathrm{C0}}(pid^0_1, pid^0_2, sa^0_1, sa^0_2, amount^0) & \\ \longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}} : & \\ & \quad \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} & \\ & \wedge \quad \mathit{non\text{-}interf\text{-}dev'}(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) & \\ & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) & \\ & \wedge \quad \mathit{impl\text{-}inv}_{\mathrm{kernel}}(\pi_{\mathrm{AK}}, c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}.cpu) & \\ & \wedge \quad \mathcal{B}(\mathit{copy}(ups, pid^0_1, pid^0_2, sa^0_1, sa^0_2, amount^0), c'_{\mathrm{ISA+DS}}) & \\ & \wedge \quad \mathit{val}_{sr}(c'_{\mathrm{ISA+DS}}) = \mathit{val}_{sr}(c_{\mathrm{ISA+DS}}) & \\ & \wedge \quad \mathit{abs\text{-}kern\text{-}unch}_{\mathrm{GM,HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) & \\ & \wedge \quad c'_{\mathrm{C0}} \in C^9_{\mathrm{C0}}. & \end{array}$$

Isabelle: `cvm/cvm_copy/cvm_copy_correct.cvm_copy_isa_correct`

Chapter 10

Verifying User Steps

In this chapter we present the last part of the CVM correctness proof accomplished in the frame of this thesis. In the previous chapter we discussed significant parts of the kernel's code verification. Besides correctness proofs of the abstract kernel step and device-communicating primitives it remains to show correctness of user computations for a complete correctness proof of the CVM model.

As described in Chapter 5 user processes are modeled as virtual assembly machines in the CVM model. This means that user processes have an illusion of their own, large, and isolated memory. Memory virtualization is transparent to user processes: within the CVM model page faults that might occur during a user step are handled silently by the low-level kernel functionality such that user can continue its run. During a single user step up to two page faults might occur: a page fault on fetch (which we also call an instruction page fault), and a page fault on load/store (to which we also refer as a data page fault). The second page fault could take place if we execute a memory operation. Our page fault is designed in a way that it guarantees that no more than two page faults occur while processing a single instruction. The following five situations are possible regarding page faults:

1. there are no page faults,
2. there is only an instruction page fault,
3. there is only a data page fault,
4. there is an instruction page fault followed by a data page fault, and
5. there is a data page fault followed by an instruction page fault.

Diagram 10.1 depicts all these cases.

Note, that on the hardware side a page fault is raised also in case of a page table length exception. This exception is not handled silently by the page-fault handler of CVM, but rather treated as a normal interrupt which triggers an execution of the abstract kernel. We distinguish two kinds of PTL exceptions: instruction and data. An instruction page table length exception takes place when the page index of the delayed

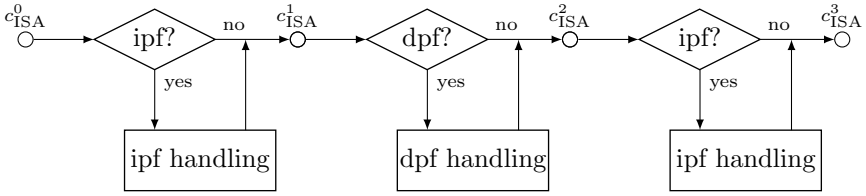

Figure 10.1: Verification diagram of user page fault handling.

program counter *dpc* exceeds the amount of process's virtual memory, i.e., the value stored in the page table length register:

$$\mathit{iptle}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \langle c_{\mathrm{ISA}}.\mathit{dpc}[31,12]\rangle > \langle c_{\mathrm{ISA}}.\mathit{spr}(\mathit{ptl})[19,0]\rangle.$$

A data page table length exception occurs when the page index of the effective address of a memory access points outside the virtual memory of the process, i.e., it is greater than the value stored in the page table length register:

$$\mathit{dptle}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=} \langle \mathit{ea}(c_{\mathrm{ISA}})[31,12]\rangle > \langle c_{\mathrm{ISA}}.\mathit{spr}(\mathit{ptl})[19,0]\rangle$$

In Figure 10.1 configurations c^0_{ISA}, c^1_{ISA}, c^2_{ISA}, and c^3_{ISA} are mapped to a single user process state at which the user attempts to make a step. In state c^1_{ISA} it is guaranteed that there is no instruction page fault (except for an instruction PTL exception *iptle*). In state c^2_{ISA} it is guaranteed that there is no data page fault (except for a data PTL exception *dptle*). In case there was an instruction page fault we also can state its absence at this point. However, if there was no instruction page fault, then one could happen during the handling of the data page fault, which will be signaled in the next step. Hence, generally we could not claim anything about the instruction page fault. Only in state c^3_{ISA} it is guaranteed that there are no instruction and data page faults.

Let us express this scenario as a formal lemma. One important fact has to be kept in mind: at the end we reach an ISA configuration such that the next step will be preformed by the processor. We need it in order to accumulate all the devices steps before the user step and after, and then later on map them to the layer of the CVM model.

Lemma 10.1 (Handling of user page faults) Assume that for an ISA configuration $c^0_{\mathrm{ISA+DS}}$ (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed, (iii) the hard disk validity properties hold, (iv) the user implementation invariants hold, (v) the $\mathcal{B}$-relation holds, and (vi) there is a sufficient amount of heap memory, then there exists a number of steps T whose execution brings ISA with devices into a configuration $c^3_{\mathrm{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk properties are preserved, (iii) the user implementation invariants are preserved, (iv) the $\mathcal{B}$-relation is preserved, (v) the value of the variable `SR` remain unchanged, (vi) it is a processor's turn to make a step, and (vii) page fault predicates may hold for the configuration $c^3_{\mathrm{ISA+DS}}$

only because of a PTL exception:

$$
\begin{array}{lll}
 & & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \\
\wedge & & \mathit{seq}\surd(\mathit{seq}, c^0_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & & \mathit{is\text{-}HD}\surd(c^0_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & & \mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c^0_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{pid}) \\
\wedge & & \mathcal{B}(\mathit{ups}, c^0_{\mathrm{ISA+DS}}) \\
\wedge & & \mathit{avail}_{\mathrm{heap}}(c_{\mathrm{C0}}) \\
\longrightarrow & & \exists\, T, c^3_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}} : \\
 & & \quad \delta^T_{\mathrm{ISA+DS}}(c^0_{\mathrm{ISA+DS}}, \mathit{seq}) = c^3_{\mathrm{ISA+DS}} \\
 & \wedge & \quad \mathit{non\text{-}interf\text{-}dev}'(c^0_{\mathrm{ISA+DS}}.\mathit{devs}, c^3_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge & \quad \mathit{is\text{-}HD}\surd(c^3_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & \wedge & \quad \mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c'_{\mathrm{C0}}, c^3_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{pid}) \\
 & \wedge & \quad \mathcal{B}(\mathit{ups}, c^3_{\mathrm{ISA+DS}}) \\
 & \wedge & \quad \mathit{val}_{\mathit{sr}}(c^3_{\mathrm{ISA+DS}}) = \mathit{val}_{\mathit{sr}}(c^0_{\mathrm{ISA+DS}}) \\
 & \wedge & \quad \mathit{abs\text{-}kern\text{-}unch}_{\mathrm{GM,HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) \\
 & \wedge & \quad \mathit{seq}(T) = \mathsf{Proc} \\
 & \wedge & \quad \mathit{is\text{-}pff}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \longrightarrow \mathit{iptle}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
 & \wedge & \quad \mathit{is\text{-}pfls}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \longrightarrow \mathit{dptle}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}).
\end{array}
$$

Isabelle: `cvm/UserStep/ipf_dpf_correct.pf_correct_isa'`

Proof. Essentially, the proof boils down to a triple application of Theorem 9.14 which concludes crucial properties of page-fault handling like not evicting the most recently swapped in page. If an instruction page fault takes place we apply the theorem and get the corresponding page loaded (in the case that no page table length exception occurs):

$$\mathit{is\text{-}pff}(c^1_{\mathrm{ISA}}) \longrightarrow \mathit{iptle}(c^1_{\mathrm{ISA}}).$$

Note, that handling of a page fault does not change the program counter and page table length values, so $\mathit{iptle}(c^i_{\mathrm{ISA}})$ will be the same for all configurations $i \in \{0..3\}$. The property that the page addressed by the program counter survives the next page fault handling, as well as the conjunct about *not next victim* addresses are ignored. For the following data page fault the same theorem will be applied and we get:

$$\mathit{is\text{-}pfls}(c^2_{\mathrm{ISA}}) \longrightarrow \mathit{dptle}(c^2_{\mathrm{ISA}}).$$

As mentioned above, we lose information about the handled instruction page fault. Thus, again ignore this conjunct, but keep in mind that

$$\mathit{not\text{-}next\text{-}victim}(c_{\mathrm{ISA}}, \mathit{pid}, \mathit{ea}(c^2_{\mathrm{ISA}})).$$

After the third application of the theorem we regain the absence of the instruction page fault and for the effective address we can conclude that the page is still in the physical memory. After that we execute the combined VAMP ISA machine up to the next sequence element which corresponds to the processor. □

The configuration $c^3_{\mathrm{ISA+DS}}$ is free of page faults. Hence, we can execute the step which corresponds to the user step in the CVM specification. The three following cases are possible:

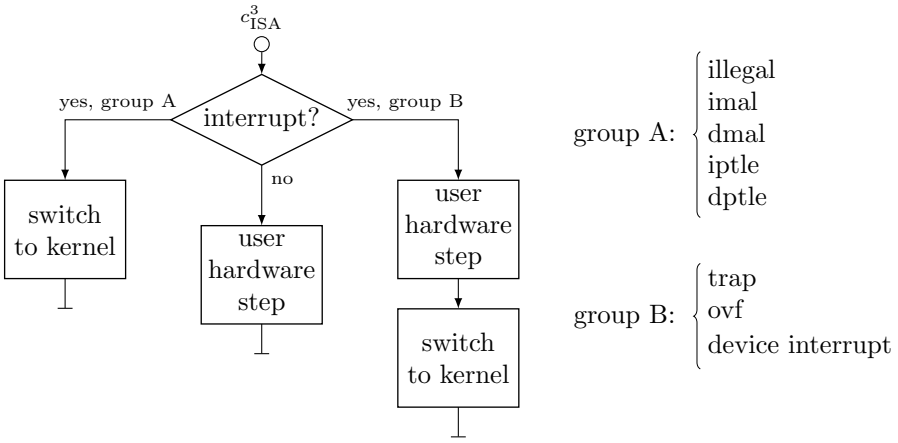

Figure 10.2: Verification diagram of a user step under assumption of page faults absence.

- no interrupts occur during the user step and the processor remains in the user mode,
- during the user step one of the devices generates an interrupt signal or the user performs the trap instruction or an arithmetic instruction with an overflow; in this case the user instruction is executed and is followed by a switch to the kernel in order to handle the interrupt with the highest level of priority,
- during an attempt to perform a step the instruction could not be executed because of a misalignment, page table length exception or an illegal interrupt; only the switch to the kernel is taken.

These three cases are reflected at Figure 10.2.

Next, we formally define predicates for the mentioned two groups of interrupts. The first group are the interrupts which abort the user execution:

$$\begin{aligned} \mathit{is\text{-}abort\text{-}intrs}(c_{\mathrm{ISA}}) \stackrel{\mathrm{def}}{=}\ & \mathit{is\text{-}ill}(c_{\mathrm{ISA}}) \\ \vee\ & \mathit{is\text{-}mal}(c_{\mathrm{ISA}}) \\ \vee\ & \mathit{iptle}(c_{\mathrm{ISA}}) \\ \vee\ & \mathit{is\text{-}iw\text{-}mem}(c_{\mathrm{ISA}}) \wedge \mathit{dptle}(c_{\mathrm{ISA}}). \end{aligned}$$

The second group of interrupts allows to execute the interrupted instruction:

$$\begin{aligned} \mathit{is\text{-}progress\text{-}intrs}(c_{\mathrm{ISA+DS}}) \stackrel{\mathrm{def}}{=}\ & \mathit{is\text{-}trap}(c_{\mathrm{ISA+DS}}.\mathit{cpu}) \\ \vee\ & c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(\mathit{sr})[6] = 1 \\ & \wedge\ \mathit{is\text{-}ovf}(c_{\mathrm{ISA+DS}}.\mathit{cpu}) \\ \vee\ & \exists\, i < 8 : \\ & c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}(\mathit{sr})[11+i] = 1 \\ & \wedge\ \mathit{intr\text{-}dev\text{-}bv}(c_{\mathrm{ISA+DS}}.\mathit{devs})[i] = 1. \end{aligned}$$

Note that, overflows and external interrupts are maskable and, therefore, they are checked together with the corresponding bit masks. Further, we introduce three separate lemmas for each case.

Lemma 10.2 (Correctness of a user step without interrupts) Assume that for an ISA configuration $c^3_{\mathrm{ISA+DS}}$ (i) the properties of the abstract kernel hold for π_{AK}, (ii) the hard disk validity properties hold, (iii) the user implementation invariants hold, (iv) the $\mathcal{B}$-relation holds, (v) there is a sufficient amount of heap memory, (vi) according to the execution sequence *seq* the next step is to be taken by the processor, (vii) there are no interrupts, and (viii) page fault predicates may hold only because of a PTL exception, then there exists a number of steps T whose execution brings ISA with devices into a configuration $c'_{\mathrm{ISA+DS}}$, such that (i) devices non-interference holds, (ii) the hard disk properties are preserved, (iii) the user implementation invariants are preserved, (iv) the $\mathcal{B}$-relation holds between the user process configuration where the user *pid* makes one step and the updated ISA configuration, and (v) the value of the variable `SR` remains unchanged:

$$
\begin{array}{ll}
 & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \\
\wedge & \mathit{is\text{-}HD}\surd(c^3_{\mathrm{ISA+DS}}.\mathit{devs}) \\
\wedge & \mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c^3_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{pid}) \\
\wedge & \mathcal{B}(\mathit{ups}, c^3_{\mathrm{ISA+DS}}) \\
\wedge & \mathit{avail}_{\mathrm{heap}}(c_{\mathrm{C0}}) \\
\wedge & \mathit{seq}(T) = \mathsf{Proc} \\
\wedge & \neg \mathit{is\text{-}abort\text{-}intrs}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \ \wedge \ \neg \mathit{is\text{-}progress\text{-}intrs}(c^3_{\mathrm{ISA+DS}}) \\
\wedge & \mathit{ipf}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \ \longrightarrow \ \mathit{iptle}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
\wedge & \mathit{dpf}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \ \longrightarrow \ \mathit{dptle}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
\longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}} : \\
 & \quad \delta^T_{\mathrm{ISA+DS}}(c^3_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev}(c^3_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & \wedge \quad \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & \wedge \quad \mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{pid}) \\
 & \wedge \quad \mathcal{B}(\mathit{one\text{-}user\text{-}step}(\mathit{ups}, \mathit{pid}), c'_{\mathrm{ISA+DS}}) \\
 & \wedge \quad \mathit{val}_{sr}(c'_{\mathrm{ISA+DS}}) = \mathit{val}_{sr}(c^3_{\mathrm{ISA+DS}})
\end{array}
$$

Isabelle: `cvm/UserStep/user_no_intr_step.user_no_intr_isa_correct`

Proof. Assumptions about interrupts allow us to claim that the jump to the interrupt service routine signal is off. Hence, we do only one step on the ISA level and instantiate T with 1. The main effort of the proof is to show that all user processes have separate memory spaces: an execution of one process does not change the state of the others. Here we use again the property proved with the help of [105, Lemma 7.22]. The relation for the process itself is similar to the simulation between ISA and assembly machines (cf. Theorem 4.8) because users are modeled by assembly machine in CVM. □

When considering a user step with an interrupt from the abort group we claim that the execution of ISA machine will lead to a state where the relation $\mathcal{B}$ holds with the original user process configuration from the lemma's assumptions. However, we enter the kernel at the point where the abstract kernel is called. Parameters of the call correspond to the values of hardware registers. This is formalized in the following postcondition:

$$
\begin{array}{lcl}
\mathit{intrPOST}(c_{\mathrm{C0}}, \mathit{eca}, \mathit{edpc}, \mathit{edata}) & \stackrel{\mathrm{def}}{=} & c_{\mathrm{C0}}.\mathit{prog} = \texttt{intr-prog} \\
& \wedge & |c_{\mathrm{C0}}.\mathit{mem.lm}| = 2 \\
& \wedge & \| \mathit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_eca}) \| = \mathit{nat}(\langle \mathit{eca} \rangle) \\
& \wedge & \| \mathit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_edata}) \| = \mathit{nat}(\langle \mathit{edata} \rangle) \\
& \wedge & \| \mathit{gvar}_{\mathrm{lm}}(1, \texttt{dispatcher\_edpc}) \| = \mathit{nat}(\langle \mathit{edpc} \rangle) \\
& \wedge & \mathit{eca} \bmod 2 = 0 \ \wedge \ 0 < \mathit{eca}/2,
\end{array}
$$

where `intr-prog` is a formally defined body of the function `dispatcher`() starting from line 35 of Listing 9.3.

Lemma 10.3 (Correctness of a user step with an abort interrupt) Assume that for an ISA configuration $c^3_{\mathrm{ISA+DS}}$ (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed and the next step is to be taken by the processor, (iii) the hard disk validity properties hold, (iv) the user implementation invariants hold, (v) the $\mathcal{B}$-relation holds, (vi) there is sufficient amount of the heap memory, (vii) there is an interrupt from the abort group, (viii) page fault predicates may hold only because of a PTL exception, then there exists a number of steps T whose execution brings ISA with deices into a configuration $c'_{\mathrm{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk properties are preserved, (iii) the kernel implementation invariants hold, (iv) the $\mathcal{B}$-relation is preserved, (v) the value of the variable `SR` remains unchanged, (vi) all global and heap variables of the abstract kernel remain unchanged, and (vii) the postcondition of the case holds:

$$
\begin{array}{lcl}
 & & \mathit{abs\text{-}kernel\text{-}props}(\pi_{\mathrm{AK}}) \\
 & \wedge & \mathit{seq}\surd(\mathit{seq}, c^3_{\mathrm{ISA+DS}}.\mathit{devs}) \ \wedge \ \mathit{seq}(T) = \mathit{Proc} \\
 & \wedge & \mathit{is\text{-}HD}\surd(c^3_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & \wedge & \mathit{impl\text{-}inv}_{\mathrm{user}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c^3_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{pid}) \\
 & \wedge & \mathcal{B}(\mathit{ups}, c^3_{\mathrm{ISA+DS}}) \\
 & \wedge & \mathit{avail}_{\mathrm{heap}}(c_{\mathrm{C0}}) \\
 & \wedge & \mathit{is\text{-}abort\text{-}intrs}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
 & \wedge & \mathit{is\text{-}pff}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \ \longrightarrow \ \mathit{iptle}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
 & \wedge & \mathit{is\text{-}pfls}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \ \longrightarrow \ \mathit{dptle}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
\longrightarrow & & \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{C0}} : \\
 & & \quad \delta^T_{\mathrm{ISA+DS}}(c^3_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} \\
 & & \wedge \ \mathit{non\text{-}interf\text{-}dev}'(c^3_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T) \\
 & & \wedge \ \mathit{is\text{-}HD}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs}) \\
 & & \wedge \ \mathit{impl\text{-}inv}_{\mathrm{kernel}}(\pi_{\mathrm{AK}}, c'_{\mathrm{C0}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu}) \\
 & & \wedge \ \mathcal{B}(\mathit{ups}, c'_{\mathrm{ISA+DS}}) \\
 & & \wedge \ \mathit{val}_{sr}(c'_{\mathrm{ISA+DS}}) = \mathit{val}_{sr}(c^3_{\mathrm{ISA+DS}}) \\
 & & \wedge \ \mathit{abs\text{-}kern\text{-}unch}_{\mathrm{GM,HM}}(\pi_{\mathrm{AK}}, c_{\mathrm{C0}}, c'_{\mathrm{C0}}) \\
 & & \wedge \ \mathit{intrPOST}(c'_{\mathrm{C0}}, \mathit{mca}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{intr\text{-}dev\text{-}bv}'(c^3_{\mathrm{ISA+DS}}.\mathit{devs})), \\
 & & \qquad\qquad c^3_{\mathrm{ISA+DS}}.\mathit{cpu.dpc}, \mathit{edata}(c^3_{\mathrm{ISA+DS}}.\mathit{cpu})).
\end{array}
$$

Isabelle: `cvm/UserStep/user_intr_no_step.user_abort_isa_correct`

Proof. We use the theorem about context switch for the case when interrupts are not handled by the page-fault handler (cf. Theorem 9.11). This theorem requires an ISA state on which the JISR effect is undone. Justifying this assumption constitutes the main effort of the proof. □

For the remaining case we assume that only external interrupts, trap, or overflow may occur. This allows the user to make a step. Nevertheless, the control is passed to the kernel. Therefore, in the lemma's conclusion the vector of user processes is updated and the kernel is in the state ready to call the abstract kernel.

Lemma 10.4 (Correctness of a user step with a continue interrupt) Assume that for an ISA configuration $c^3_{\text{ISA+DS}}$ (i) the properties of the abstract kernel hold for π_{AK}, (ii) the execution sequence *seq* is well-formed and the next step is to be taken by the processor, (iii) the hard disk validity properties hold, (iv) the user implementation invariants hold, (v) the $\mathcal{B}$-relation holds, (vi) there is sufficient amount of the heap memory, (vii) there are no interrupt from the abort group, (viii) there is an external, trap, or overflow interrupt, (ix) page fault predicates may hold only because of a PTL exception, then there exists a number of steps T whose execution brings ISA with deices into a configuration $c'_{\text{ISA+DS}}$ and a C0 configuration c'_{C0}, such that (i) the devices other than the hard disk do not interfere with the processor, (ii) the hard disk properties are preserved, (iii) the kernel implementation invariants hold, (iv) the $\mathcal{B}$-relation holds between the user process configuration where the user *pid* makes one step and the updated ISA configuration, (v) the value of the variable `SR` remain unchanged, (vi) all global and heap variables of the abstract kernel remain unchanged, and (vii) the postcondition of the case hold (in this case we use *pc* instead of *dpc*):

$$
\begin{array}{lll}
 & \textit{abs-kernel-props}(\pi_{\text{AK}}) & \\
\wedge & \textit{seq}\surd(\textit{seq}, c^3_{\text{ISA+DS}}.\textit{devs}) \ \wedge \ \textit{seq}(T) = \textit{Proc} & \\
\wedge & \textit{is-HD}\surd(c^3_{\text{ISA+DS}}.\textit{devs}) & \\
\wedge & \textit{impl-inv}_{\text{user}}(\pi_{\text{AK}}, c_{\text{C0}}, c^3_{\text{ISA+DS}}.\textit{cpu}, \textit{pid}) & \\
\wedge & \mathcal{B}(\textit{ups}, c^3_{\text{ISA+DS}}) & \\
\wedge & \textit{avail}_{\text{heap}}(c_{\text{C0}}) & \\
\wedge & \neg\textit{is-abort-intrs}(c^3_{\text{ISA+DS}}.\textit{cpu}) \ \wedge \ \textit{is-progress-intrs}(c^3_{\text{ISA+DS}}) & \\
\wedge & \textit{is-pff}(c^3_{\text{ISA+DS}}.\textit{cpu}) \ \longrightarrow \ \textit{iptle}(c^3_{\text{ISA+DS}}.\textit{cpu}) & \\
\wedge & \textit{is-pfls}(c^3_{\text{ISA+DS}}.\textit{cpu}) \ \longrightarrow \ \textit{dptle}(c^3_{\text{ISA+DS}}.\textit{cpu}) & \\
\longrightarrow & \exists\, T, c'_{\text{ISA+DS}}, c'_{\text{C0}} : & \\
 & \quad \delta^T_{\text{ISA+DS}}(c^3_{\text{ISA+DS}}, \textit{seq}) = c'_{\text{ISA+DS}} & \\
 & \wedge \quad \textit{non-interf-dev}'(c^3_{\text{ISA+DS}}.\textit{devs}, c'_{\text{ISA+DS}}.\textit{devs}, \textit{seq}, T) & \\
 & \wedge \quad \textit{is-HD}\surd(c'_{\text{ISA+DS}}.\textit{devs}) & \\
 & \wedge \quad \textit{impl-inv}_{\text{kernel}}(\pi_{\text{AK}}, c'_{\text{C0}}, c'_{\text{ISA+DS}}.\textit{cpu}) & \\
 & \wedge \quad \mathcal{B}(\textit{one-user-step}(\textit{ups}, \textit{pid}), c'_{\text{ISA+DS}}) & \\
 & \wedge \quad \textit{val}_{sr}(c'_{\text{ISA+DS}}) = \textit{val}_{sr}(c^3_{\text{ISA+DS}}) & \\
 & \wedge \quad \textit{abs-kern-unch}_{\text{GM,HM}}(\pi_{\text{AK}}, c_{\text{C0}}, c'_{\text{C0}}) & \\
 & \wedge \quad \textit{intrPOST}(c'_{\text{C0}}, \textit{mca}(c^3_{\text{ISA+DS}}.\textit{cpu}, \textit{intr-dev-bv}'(c^3_{\text{ISA+DS}}.\textit{devs})), & \\
 & \qquad\qquad c^3_{\text{ISA+DS}}.\textit{cpu.pc}, \textit{edata}(c^3_{\text{ISA+DS}}.\textit{cpu})). &
\end{array}
$$

Isabelle: `cvm/UserStep/user_intr_step.user_prog_dev_intr_isa_correct`

Proof. The case might be seen as a combination of two previous cases, therefore, we reuse significant parts of the proofs of lemmas 10.2 and 10.3. □

Chapter 11

Summary and Future Work

In this thesis we have extended the Verisoft technology for verification of C programs to handle realistic systems implementations featuring inline assembly portions and have applied the developed methodology to prove CVM, a framework for microkernel programmers, correct. The distinctive feature of the work is pervasiveness: the thesis presents to the best of our knowledge the first formal correctness proof of a microkernel programmed in C with inline assembly running concurrently with user processes on verified hardware. The hardware constitutes a processor model on the level of instruction set architecture and models of different devices.

In order to achieve the results of the thesis the following has been accomplished.

- CVM, a formal computation model for concurrent user processes interacting with a generic microkernel and devices has been defined in Isabelle in collaboration with many colleagues from the Verisoft project. Besides Isabelle, the model was implemented in C0, a slightly restricted dialect of C, with inline assembly as a framework featuring virtual memory support, demand paging, memory management, and low-level inter-process and devices communications.

- The formal definition of CVM requires to import various computational models. We have used C0 small-step semantics to model kernel computations. We have instantiated a general framework of combined systems with an ISA model of the verified processor VAMP and models of several devices including a hard disk. The C0 semantics was formally connected with the ISA semantics by introducing an intermediate level: the VAMP assembly model. Correctness of the latter towards VAMP ISA has been proven in two flavors: with and without devices access.

- In collaboration with In der Rieden we have developed a formal theory of linking C0 programs in order to support separate kernel modules: the low-level kernel functionality implemented with inline assembly portions is verified separately from the upper layers of the kernel. Correctness of the linker was shown completely in the frame of this thesis.

- We have stated the overall correctness theorem of CVM which justifies concurrent executions of user processes with a kernel and devices on a hardware model. We

have proven this theorem for the cases of context switching, primitive `cvm_copy`(), and user steps.

- During verification of these cases we have to reason about source code containing inline assembly portions. For this we have developed formal inline assembly semantics.
- We have imported a formal theory of a verified page-fault handler and have proven liveness of a double call to the handler.

Based on the CVM framework two microkernels have been built, tested, and verified to a large extent in Verisoft:

- VAMOS, an L4-inspired general purpose microkernel which provides a process scheduler, an infrastructure for communication with hardware devices, and message passing between processes, and
- OLOS, an OSEKtime-like operating system, used in a distributed automotive real-time system establishing eCall functionality.

Pervasive correctness proofs for both microkernels, i.e., justification that their C0 implementations have intended behavior on the ISA level, will require application of the CVM top-level correctness theorem.

Formal theories in Isabelle developed in the scope of this work comprise at least 3100 definitions and 5500 lemmas proven in up to 86000 steps.

Finally, we point out possible directions of future work.

Verification of primitives. In this thesis we have developed and successfully applied an approach to verification of CVM primitives. However, quite a few primitives still have to be verified. The most interesting case constitute primitives accessing devices. Their correctness proofs seem to be tricky because they will require formulation of additional invariants about devices and adding restrictions over execution sequences in the CVM top-level theorem in order to reorder multiple device accesses. Note that a similar problem — verification of a hard disk driver — is shown by Alkassar in [5].

Abstract kernel step correctness. As yet this case remains unproven in the CVM top-level theorem. The thesis of In der Rieden [32] presents a C0-level simulation proof between a linked program consisting of two modules and one of these modules. This proof assumes particular properties of a compiler, namely the ability of the compiler to allocate specified functions and variables at addresses provided by the user. However, the current version of the Verisoft's C0 compiler [66] does not respect this property. In spite of this, a larger part of the mentioned proof could be used in the context of the abstract kernel step correctness.

Altogether, the following goals have to be shown in order to finish the abstract kernel step correctness proof.

- The cases of an abstract kernel invocation and return have to be considered. The letter includes a proof of the relation for abstract kernel executions results (Definition 7.9).
- The correctness of the overall abstract kernel step, including the two aforementioned cases, has to be transfered to the VAMP ISA level.

- Provided that the two above points are accomplished, we obtain a proof that the abstract kernel relation is preserved under abstract kernel steps. However, this is just a part of a complete CVM abstraction relation (cf. Section 7.1.5). Its remaining parts — the relations for user processes, status register, and devices — as well as the CVM implementation invariants (Section 7.1.6) have to be proven preserved under abstract kernel steps.
- Finally, the lemma of the abstract kernel step correctness must be applied in the context of the CVM induction step proof. This includes appropriate instantiation of the CVM sequence as well as the CVM steps number.

Waiting for interrupts correctness. This models the kernel idle state and is currently unproven. The desired proof includes a C0-invocation of the function `cvm_wait` and showing the correctness of its body, an endless assembly loop implemented only with two instructions. Likewise the case of an abstract kernel step, the proof has to be transfered to the VAMP ISA level. A proof that the CVM abstraction relation and implementation invariants are preserved under the function is relatively easy since none of its assembly instructions writes the memory.

Proof automation. Although we used mostly interactive verification techniques there is room for automation. One can gain from methods of automated verification while proving functional correctness of the source code. We used the ML code generation mechanism for the proof of the microkernel source code well-formedness properties required by the C0 compiler correctness theorem. That saved several thousands of proof commands. The next possible candidates for proof automation are assembly portions. Due to the relatively simple finite memory model it might be possible to obtain the values of desired memory cells by means of model checking. In order to ease the C0 part verification, one can think of a Hoare logic environment for the C0 small step semantics which will automatically generate verification conditions to be proven.

Bibliography

[1] Programming languages — c. International Standard ISO/IEC ISO/IEC 9899:1999, 1999.

[2] ISO/IEC 15408. Common criteria for information technology security evaluation. Available at `http://www.commoncriteriaportal.org`, 1999.

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

[4] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. In *Journal of Automated Reasoning: Special Issue on Operating Systems Verification*. Springer, 2009.

[5] Eyad Alkassar. *OS Verification Extended. On the Formal Verification of Device Drivers and the Correctness of Client/Server Software.* PhD thesis, Saarland University, Computer Science Department, 2009.

[6] Eyad Alkassar, Peter Böhm, and Steffen Knapp. Formal correctness of a gate-level automotive bus controller implementation. In *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES08)*. Springer Science and Business Media, 2008.

[7] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In Natarajan Shankar and Jim Woodcock, editors, *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 225–239. Springer, 2008.

[8] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In Juris Hartmanis Gerhard Goos and Jan van Leeuwen, editors, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[9] National ICT Australia. Website. `http://www.nicta.com.au`, 2008.

[10] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*. Springer, 2006.

[11] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *LNCS*. Springer, 2005.

[12] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual : Version 6.1*. INRIA, 1997.

[13] Christoph Baumann and Thorsten Bormer. Verifying the PikeOS microkernel: An overview of the Verisoft XT avionics project. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009. To appear.

[14] Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification — experiences from the verisoft email client. In *Proceedings of the Workshop on Empirical Succesfully Computerized Reasoning (ESCoR 2006)*, 2006.

[15] W. R. Bevier. *A Verified Operating System Kernel*. PhD thesis, University of Texas at Austin, 1987.

[16] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[17] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An approach to systems verification. 5(4):411–428, December 1989.

[18] W.R. Bevier, W.A. Hunt, J. Strother Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[19] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, March 2005.

[20] 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.

[21] Sebastian Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Computer Science Department, 2008.

[22] Robert S. Boyer and J. Strother Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[23] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 120–133, New York, NY, USA, 1993. ACM.

[24] Inc. Computational Logic. Website. `http://www.computationallogic.com`, 2008.

[25] Intel Corporation. Intel virtualization technology website. `http://www.intel.com/technology/virtualization/`, 2008.

[26] Coyotos. Website. `http://www.coyotos.org`, 2008.

[27] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Computer Science Department, July 2006.

[28] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725, pages 301–316. Springer, 2005.

[29] Matthias Daum. Modelling user programs on top of a microkernel. In *Doctoral Symposium at FM'08, technical report*. Turku centre for computer science, 2008.

[30] Matthias Daum. *To appear.* PhD thesis, Saarland University, Computer Science Department, 2009.

[31] Matthias Daum, Jan Drenbher, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Journal of Automated Reasoning: Special Issue on Operating System Verification*. Springer, 2009.

[32] Tomas In der Rieden. *Verified Linking for Modular Kernel Verification.* PhD thesis, Saarland University, Computer Science Department.

[33] J. Dörenbächer. Vamos microkernel: Formal models and verification. A talk given at *Intl Workshop on System Verification,*. `www.cse.unsw.edu.au/formalmethods/events/svws-06/VAMOS_Microkernel.pdf`, 2006.

[34] Jan Dörenbächer. *To appear.* PhD thesis, Saarland University, Computer Science Department, 2009.

[35] Endrawaty. Verification of the fiasco IPC implementation, 2005.

[36] Galen C. Hunt et al. *An Overview of the Singularity Project. Microsoft Research Technical Report MSR-TR-2005-135*. Microsoft Corporation, 2005.

[37] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[38] R. Feiertag and P. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the National Computer Conference 48*, pages 329–334, 1979.

[39] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, New York, NY, USA, June 2008. ACM.

[40] Arthur David Flatau. *A verified implementation of an applicative language with dynamic storage allocation*. PhD thesis, Austin, TX, USA, 1992.

[41] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603, pages 1–16. Springer, 2005.

[42] D.I. Good, R.L. London, and W.W. Bledsoe. An interactive program verification system. *IEEE Transactions On Software Engineering SE-1*, 1:59–67, March 1975.

[43] Mike Gordon. From lcf to hol: a short history. pages 169–185, 2000.

[44] P. Brinch Hansen. The Nucleus of multiprogramming operating systems. *Communications of ACM*, 13:238–250, 1970.

[45] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The nizza secure-system architecture. In *1st International Conference on Collaborative Computing: Networking, Applications, and Worksharing*, San Jose, CA, USA, 2005. IEEE.

[46] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S.M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *Operating Systems Review*, 41(4):3–11, 2007.

[47] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[48] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[49] M. A. Hillebrand and W. J. Paul. On the architecture of system verification environments. In *Haifa Verification Conference 2007, October 23-25, 2007, Haifa, Israel*, LNCS. Springer, 2007.

[50] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Computer Science Department, June 2005.

[51] 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.

[52] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[53] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. Technical Report No. 187 Institut für Informatik Universität Freiburg, url = citeseer.ist.psu.edu/article/hohmuth03semantics.html.

[54] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *2nd ECOOP Workshop on Program Languages and Operating Systems (ECOOP-PLOS'05)*, 2005.

[55] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[56] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[57] T. In der Rieden and A. Tsyban. Cvm - a verified framework for microkernel programmers. In *3rd International Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[58] SRI International. Website. `http://www.sri.com`, 2008.

[59] E. Northup S. Sridhar J. Shapiro, M. S. Doerrie and M. Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification*, October 2004.

[60] Dave Jaggar. Arm architecture and systems. *IEEE Micro*, 17(4):9–11, 1997.

[61] Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.

[62] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[63] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[64] Steffen Knapp. *Pervasive Verification of Distributed Real-Time Systems*. PhD thesis, Saarland University, Computer Science Department.

[65] D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[66] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Department, 2007.

[67] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Bernhard Aichernig and Bernhard Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5–9 September 2005, Koblenz, Germany*, pages 2–11, 2005.

[68] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. *SIGOPS Oper. Syst. Rev.*, 9(5):132–140, 1975.

[69] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles (SOSP-14)*.

[70] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[71] Jochen Liedtke. Microkernels must and can be small. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Seattle, WA, October 1996.

[72] Jochen Liedtke. Towards real microkernels. 39(9):70–77, September 1996.

[73] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The l4ka vision, April 2001.

[74] Microsoft Corp. Verisoft - formal verification of computer systems. `http://www.microsoft.com/emic/verisoft.mspx`, 2007.

[75] Microsoft Corp. Vcc: A c verifier. `http://research.microsoft.com/en-us/projects/vcc`, 2008.

[76] Microsoft Corp. Windows server 2008 virtualization with hyper-v. `http://www.microsoft.com/windowsserver2008/en/us/hyperv.aspx`, 2008.

[77] J S. Moore. Piton: A mechanically verified assembly-level language. *Automated Reasoning Series*, 1996.

[78] 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, pages 161–172. Springer, 2003.

[79] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[80] P. Neumann and R. Feiertag. PSOS revisited. In *19th Annual Computer Security Applications Conference*, 2003.

[81] P.G. Neumann, R.S Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. *A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116*. Computer Science Laboratory, SRI International, Menlo Park, California, May 1980.

[82] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 320–333, New York, NY, USA, 2006. ACM.

[83] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using xcap to certify realistic systems code: Machine context management. In *TPHOLs*, pages 189–206, 2007.

[84] Radboud University Nijmegen. Website. `http://www.ru.nl`, 2008.

[85] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002.

[86] University of California at Los Angeles. Website. http://www.ucla.edu, 2008.

[87] Technical University of Dresden. Website. http://www.tu-dresden.de, 2008.

[88] University of Texas at Austin. Website. http://www.utexas.edu, 2008.

[89] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[90] Wolfgang Paul. Towards a worldwide verification technology. In *Verified Software: Theories, Tools, Experiments (VSTTE 2005), Proceedings*, Zürich, Switzerland, October 2005.

[91] Wolfgang Paul. System architecture. lecture notes. http://busserver.cs.uni-sb.de/lehre/vorlesung/info2/ss08/material/mitschrift07.pdf, 2007.

[92] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[93] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, May 2007.

[94] Robin Project. Website. http://robin.tudos.org, 2008.

[95] The L4Ka Project. Website. http://l4ka.org, 2008.

[96] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, and Robert Baron. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *SIGARCH Comput. Archit. News*, 15(5):31–39, 1987.

[97] Verisoft Repository. Website. http://www.verisoft.de/VerisoftRepository.html, 2009.

[98] Microsoft Research. Website. http://research.microsoft.com, 2008.

[99] L. Robinson and O. Roubine. *Special - A Specification and Assertion Language. Technical Report CSL-46*. Stanford Research Institute, Menlo Park, California, January 1977.

[100] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.

[101] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, volume 3452, pages 398–414. Springer, 2005.

[102] Andrey Shadrin. Design and implementation of the portmapper and rpc primitives in the context of the sos. Master's thesis, Saarland University, 2006.

[103] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *17th ACM Symposium on Operating Systems Principles*, pages 170–185, December 1999.

[104] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *IEEE Symposium on Security and Privacy*, pages 166–176, May 2000.

[105] Artem Starostin. *Formal Verification of Demand Paging. To appear.* PhD thesis, Saarland University, Computer Science Department.

[106] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In *3rd International Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[107] Artem Starostin and Alexandra Tsyban. Verified process-context switch for c-programmed kernels. In Natarajan Shankar and Jim Woodcock, editors, *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 240–254. Springer, 2008.

[108] SYSGO AG. Sysgo embedding innovations. `http://www.sysgo.com/`, 2007.

[109] Hendrik Tews. Verifying duff's device: A simple compositional denotational semantics for goto and computed jumps.

[110] Hendrik Tews. Micro hypervisor verification: possible approaches and relevant properties. `http://robin.tudos.org/publications/hyperveri.pdf`, 2007.

[111] The Verisoft Consortium. The Verisoft Project. `http://www.verisoft.de/`, 2008.

[112] The Verisoft XT Consortium. The Verisoft XT Project. `http://www.verisoftxt.de/`, 2009.

[113] 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.

[114] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–108, New York, NY, USA, 2007. ACM.

[115] Sergey Tverdyshev. *Formal Verification of the VAMP.* PhD thesis, Saarland University, Computer Science Department.

[116] Yale University. Website. `http://www.yale.edu`, 2008.

[117] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 64–65, New York, NY, USA, 1979. ACM.

[118] Jr. Warren A. Hunt. *FM8501: a verified microprocessor.* Springer-Verlag, London, UK, 1994.

[119] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 268–279, London, UK, 1993. Springer-Verlag.

[120] Mickey Williams. *Microsoft Visual C# .NET*. Microsoft Press, 2002.

[121] William David Young. *A verified code generator for a subset of gypsy*. PhD thesis, 1988. Supervisor-Robert S. Boyer and Supervisor-J. Strother Moore.

Index