Noname manuscript No.
(will be inserted by the editor)

Balancing the Load

Leveraging a Semantics Stack for Systems Verification

Eyad Alkassar · Mark A. Hillebrand · Dirk C. Leinenbach · Norbert W. Schirmer · Artem Starostin · Alexandra Tsyban

Received: date / Accepted: date

Abstract We have developed a stack of semantics for a high-level C-like language and low-level assembly code, which has been carefully crafted to support the pervasive verification of system software. It can handle mixed-language implementations and concurrently operating devices, and permits the transferral of properties to the target architecture while obeying its resource restrictions. We demonstrate the applicability of our framework by proving the correct virtualization of user memory in our microkernel, which implements demand paging. This verification target is of particular interest because it has a relatively simple top-level specification and it exercises all parts of our semantics stack. At the bottom level a disk driver written in assembly implements page transfers via a swap disk. A page-fault handler written in C uses the driver to implement the paging algorithm. It guarantees that a step of the currently executing user can be simulated at the architecture level. Besides the mere theoretical and technical difficulties the project also bore the social challenge to manage the large verification effort, spread over many sites and people, concurrently contributing to and maintaining a common theory corpus. We share our experiences and elaborate on lessons learned.

Keywords Pervasive formal verification · systems verification · software verification

1 Introduction

The context of this work is the German Verisoft project, a large scale effort bringing together industrial and academic partners to push the state of the art in formal verifi-

Work supported by the German Federal Ministry of Education and Research (BMBF) under grant 01 IS C38. Work of the first and the third author supported by DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". Work of the fifth author supported by the International Max Planck Research School for Computer Science (IMPRS-CS).

M. A. Hillebrand · D. C. Leinenbach · N. W. Schirmer
German Research Center for Artificial Intelligence (DFKI), P.O. Box 15 11 50, 66041 Saarbrücken, Germany
E-mail: {mah, Dirk.Leinenbach, Norbert.Schirmer}@dfki.de

E. Alkassar · A. Starostin · A. Tsyban
Saarland University, Computer Science Dept., P.O. Box 15 11 50, 66041 Saarbrücken, Germany
E-mail: {eyad, starostin, azul}@wjpserver.cs.uni-sb.de

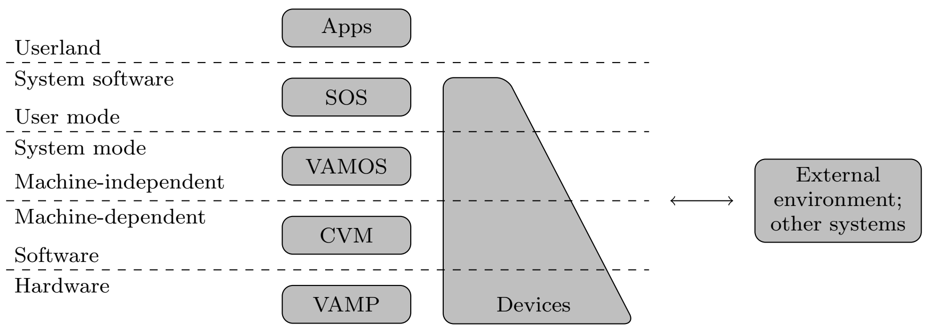


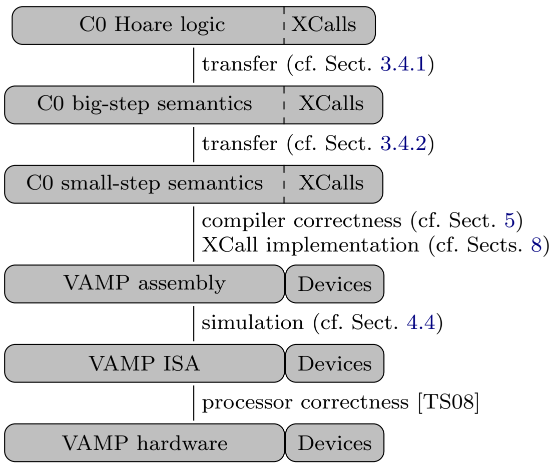
Fig. 1 Implementation layers of the academic system (Verisoft subproject 2)

cation for realistic computer systems, comprising hard- and software. In this article we present parts of Verisoft's 'academic system', which is a computer system for writing, signing, and sending emails. As it covers all implementation layers from the gate level hardware up to communicating user processes it is a representative of a vertical slice of a general-purpose computer system. We pay special attention to *pervasive* verification, which means that at the end of the day we obtain a correctness result for the actual system, which is the hardware running the system. We do not only verify isolated portions on different abstraction layers but make sure that we can combine and preserve the results for the actual system.

One key obstacle in a large formal and layered artefact as the Isabelle/HOL theories of our project is the seamless integration of the abstraction layers. This is achieved through simulation and transfer theorems, gluing together the layers. The results presented in conference articles are often simplified since they focus on a certain point and have to fit into the page limit. With this article we attempt to overcome a popular demand and present key theorems of the Verisoft system stack up to the microkernel level in great detail. We do not present all the underlying definitions. Here we refer to previous work and our repository.[1] Instead we discuss the shape of the theorems and the nature of various assumptions with regard to their role in the overall verification.

System stack. The *hardware architecture* is called VAMP, a DLX like processor that supports address translation and memory-mapped I/O devices. With the next level of *communicating virtual machines* (CVM) a hardware-independent programming interface for a microkernel is provided, that establishes the notion of separate concurrent user processes. Parts of the CVM are implemented in assembly, because C0, our main implementation language and a subset of C, lacks some low-level programming constructs. On the basis of the CVM our *microkernel* VAMOS [DDB08] is programmed in pure C0. The *simple operating system* (SOS) is implemented as a (privileged) user process of VAMOS [Bog08]. It offers file I/O, network access and inter-process communication. On top of it user applications are provided with a client / server architecture based on remote procedure calls [ABP09]. Finally these user applications implement the functionality of the academic system: signing software, SMTP client and server [LNRS07], and a simple mail user agent [BHW06]. The implementation stack is also depicted in Figure 1.

[1] The Isabelle theories of all models and proofs of this article are available in the published portions of the Verisoft repository [HP08] at http://www.verisoft.de/VerisoftRepository.html.



C0 Hoare logic
XCalls
transfer (cf. Sect. 3.4.1)
C0 big-step semantics
XCalls
transfer (cf. Sect. 3.4.2)
C0 small-step semantics
XCalls
compiler correctness (cf. Sect. 5)
XCall implementation (cf. Sects. 8)
VAMP assembly
Devices
simulation (cf. Sect. 4.4)
VAMP ISA
Devices
processor correctness [TS08]
VAMP hardware
Devices

Fig. 2 Semantics stack

The computational models we introduce to specify and verify the academic system are assembled along the implementation layers. In this article we focus on the layers up to the CVM, where the challenging aspect is the integration of concurrent computations of the processor and devices at the architecture level into the sequential view provided by the C0 (and assembly) language. We introduce the C0 semantics stack which is orthogonal to the system stack described before. With the semantics stack we establish a convenient Hoare logic to reason about the sequential parts of C0 programs (without inline assembly code) and simultaneously provide the means to compose the results to deal with assembly code and to integrate devices.

Semantics stack. The C0 semantics stack comprises a Hoare logic, a big-step semantics, and a small-step semantics. It can be continued to the VAMP machine level, which is divided further into assembly layer, instruction set architecture, and gate level hardware. An overview is depicted in Figure 2. By a higher level of abstraction in the Hoare logic compared to the small-step semantics, we gain efficiency for the verification of individual C0 programs. However, since the semantics stack is merely a proof device for C0 programs we have to integrate the results obtained in the Hoare logic to our systems stack. We supply soundness and simulation theorems that permit the transferral of program properties from the Hoare logic down to the small-step semantics. Applying compiler correctness we map those properties to assembly machines. We can get further down to the ISA layer by employing a simulation theorem and finally to the hardware by employing a processor correctness result.

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

Up to now we have argued how to bring the results down to the lower levels such that we can conduct reasoning at a comfortable abstraction level. However, this comes at the cost of expressiveness. Most prominently only the levels below C0 allow the integration of devices, which are a concurrent source of computation. As soon as we attempt to reason about C0 programs that use these devices we either have to be able to express device operations at the Hoare logic level or we are forced to carry out the whole verification at the assembly level. Our approach is to abstract the effect of those low-level computations into atomic 'XCalls' (extended calls) in all our semantic layers. The state space of C0 is augmented with an additional component that represents the state of the external component, e.g., the device. An XCall is a procedure call that makes a transition on this external state and communicates with C0 via parameter passing and return values. With this model it is straightforward to integrate XCalls into the semantics and into Hoare logic reasoning. The XCall is typically implemented in assembly. An implementation proof of this piece of assembly justifies the abstraction to an atomic XCall.

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

Related work. As Klein [Kle09] provides an excellent and comprehensive overview of the history and current state of the art in operating systems verification we limit this paragraph to highlight the peculiarities of our work. We extend the seminal work on the CLI stack [BHMY89] by integrating devices into our model and targeting a more realistic system architecture regarding both hard and software. The project L4.verified [HEK$^+$07] focuses on the verification of an efficient microkernel, rather than on formal pervasiveness, as no compiler correctness or an accurate device interaction is considered. In the FLINT project, an assembly code verification framework is developed and code for context switching on a x86 architecture was formally proven [NYS07]. A program logic for assembly code is presented, but no integration of results into high-level programming languages is undertaken. Relevant references to our own work are given at the beginning of each section.

Outline. In Section 2 we introduce Isabelle/HOL and notational conventions. In Section 3 we introduce the language C0 and the associated semantics stack. We present theorems that permit the transferral of properties from the Hoare logic down to the small-step semantics. In Section 4 we present models for the assembly language and instruction set of the VAMP architecture. We also show how to integrate devices into these models. Section 5 deals with verified compilation from C0 to VAMP. In Sections 6 to 8 previous results are employed and combined to establish a correctness result for CVM. Going top-down we first present the theorem for CVM user step correctness in Section 6. One case in the proof of this theorem is the correct handling of page-faults. In Section 7 we present the page-fault handler that implements demand paging for CVM. For paging operations it uses a disk driver, which is presented in Section 8. We wrap up our experience and 'lessons learned' in Section 9. In the appendix we provide a glossary on important constants and notations.

2 Preliminaries

The formalizations presented in this article are mechanized and checked within the generic interactive theorem prover *Isabelle* [Pau94]. Isabelle is called generic as it provides a framework to formalize various *object logics* that are declared via natural deduction style inference rules within Isabelle's meta-logic *Pure*. The object logic that we employ for our formalization is the higher order logic of *Isabelle/HOL* [NPW02].

This article is written using Isabelle's document generation facilities, which guarantees that the presented theorems correspond to formally proven ones. We distinguish formal entities typographically from other text. We use a sans serif font for types and constants (including functions and predicates), e.g., map, a slanted serif font for free variables, e.g., x, and a slanted sans serif font for bound variables, e.g., x. Small capitals are used for data type constructors, e.g., $\textsc{Foo}$, and type variables have a leading tick, e.g., $'a$. HOL keywords are typeset in type-writer font, e.g., `let`. We also take the freedom to borrow C notation, e.g., `unsigned` when presenting C0.

As Isabelle's inference kernel manipulates rules and theorems at the Pure level the meta-logic becomes visible to the user and also in this article when we present theorems and lemmas. The Pure logic itself is intuitionistic higher order logic, where universal quantification is $\bigwedge$ and implication is $\Longrightarrow$. Nested implications like $P_1 \Longrightarrow P_2 \Longrightarrow P_3 \Longrightarrow C$ are abbreviated with $[\![P_1; P_2; P_3]\!] \Longrightarrow C$, where one refers to P_1, P_2, and P_3 as the premises and to C as the conclusion. To group common premises and to support modular reasoning Isabelle provides *locales* [Bal03, Bal06].

In the object logic HOL universal quantification is $\forall$ and implication is $\longrightarrow$. The other logical and mathematical notions follow the standard notational conventions with a bias towards functional programming. We only present the more unconventional parts here. We prefer curried function application, e.g., $f\ a\ b$ instead of $f(a, b)$. In this setting the latter becomes a function application to *one* argument, which happens to be a pair.

Isabelle/HOL provides a library of standard types like Booleans, natural numbers, integers, total functions, pairs, lists, and sets and packages to define new data types and records. Isabelle allows polymorphic types, e.g., $'a\ \mathsf{list}$ is the list type with type variable $'a$. In HOL all functions are total, e.g., $\mathsf{nat} \Rightarrow \mathsf{nat}$ is a total function on natural numbers. Function update is $f(y := v) \equiv \lambda \mathsf{x}.\ \mathtt{if}\ \mathsf{x} = y\ \mathtt{then}\ v\ \mathtt{else}\ f\ \mathsf{x}$ and function composition is $f \circ g \equiv \lambda \mathsf{x}.\ f\ (g\ \mathsf{x})$. To formalize partial functions the type $'a\ \mathsf{option}$ is used. It is a data type with two constructors, one to inject values of the base type, e.g., $\lfloor x \rfloor$, and the additional element $\bot$. A base value can be projected with the function the, which is defined by the sole equation $\mathsf{the}\ \lfloor x \rfloor = x$. Since HOL is a total logic the term $\mathsf{the}\ \bot$ is still a well-defined yet un(der)specified value. Partial functions can be represented by the type $'a \Rightarrow\ 'b\ \mathsf{option}$, abbreviated as $'a \rightharpoonup\ 'b$.

The syntax and the operations for lists are similar to functional programming languages like ML or Haskell. The empty list is [], with $x \cdot xs$ the element x is 'consed' to the list xs, the head of list xs is $\mathsf{hd}\ xs$ and the remainder, its tail, is $\mathsf{tl}\ xs$. With $xs\ @\ ys$ list ys is appended to list xs. With $\mathsf{map}\ f\ xs$ the function f is applied to all elements in xs. The length of a list is $|xs|$, the n-th element of a list can be selected with $xs[n]$ and updated via $xs[n := v]$. An entry of a two-dimensional list is updated by $xs[n, m := v]$. With $\mathsf{set}\ xs$ we obtain the set of elements in list xs. Filtering those elements from a list for which predicate P holds is achieved by $[\mathsf{x}{\in}xs\ .\ P\ \mathsf{x}]$. With $\mathsf{replicate}\ n\ e$ we denote a list that consists of n elements e.

Table 1 C0 Expressions e

Lit v	literal values v
VarAcc vn	access of variable vn
ArrAcc e_a e	indexing array e_a with index e
StructAcc e cn	selecting component cn of structure e
BinOp bop e_1 e_2	binary operation
LazyBinOp $lbop$ e_1 e_2	lazy binary operation
UnOp uop e	unary operation
AddrOf e	address of (left-) expression e
Deref e	dereferencing e

Sets come along with the standard operations for union, i.e., $A \cup B$, intersection, i.e., $A \cap B$ and membership, i.e., $x \in A$. The set image f ` A yields a new set by applying function f to every element in set A.

Partial functions $'a \rightharpoonup 'b$ are commonly used as *maps*. With map-of xs we construct a map from an association list, i.e., a list of key / value pairs. We denote the domain of map m by dom m. With $m_1 ++ m_2$ we add the map m_2 to map m_1, where entries of m_1 are overwritten if necessary. We can restrict the domain of a map m to a set A by $m\restriction_A$. Subsumption of maps is defined as $m_1 \subseteq_m m_2 \equiv \forall\, a \in \mathsf{dom}\ m_1.\ m_1\ a = m_2\ a$ and composition of maps as $m_1 \circ_m m_2 \equiv \lambda k.\ \mathtt{case}\ m_2\ k\ \mathtt{of}\ \bot \Rightarrow \bot \mid \lfloor v \rfloor \Rightarrow m_1\ v$.

A record is constructed by assigning all of its fields, e.g., $(\!|\mathsf{fld}_1 = v_1,\ \mathsf{fld}_2 = v_2|\!)$. Field fld_1 of record r is selected by $r.\mathsf{fld}_1$ and updated with a value x via $r(\!|\mathsf{fld}_1 := x|\!)$.

The first and second component of a pair can be accessed with the functions fst and snd. Tuples with more than two components are pairs nested to the right.

3 C0

C0 is a type safe subset of C designed with verification in mind. An elaborate description of the big-step semantics is given in [Sch06] and the small-step semantics is introduced in [Lei08]. An overview on the simulation theorems between the semantical layers can be found in [AHL+08], the transfer from the Hoare logic to the big-step semantics is detailed in [Sch06].

The primitive values are signed and unsigned integers (32 bit), 8-bit chars, Booleans, and typed pointers. Aggregate values comprise structures and arrays. Unions and pointer arithmetic are not supported. C0 Expressions e and statements s are defined as data types (cf. Tables 1 and 2). Binary operations are arithmetic operations (`+`, `-`, `*`, `/`, `%`), bitwise operations (`|`, `&`, `^`, `<<`, `>>`), and comparisons (`>`, `<`, `==`, `!=`, `>=`, `<=`). Lazy binary operations are Boolean conjunction `&&` and disjunction `||`. Unary operations are unary minus `-`, bitwise negation `~`, logical negation `!`, and operations to convert between integral values (integers, unsigned integers, and chars). Left expressions are the subset of expressions that refer to memory objects (e.g., VarAcc, ArrAcc, StructAcc, and Deref).

Statement identifiers *sid* are used in the compiler correctness theorem. Procedures assign the return value to a (global or local) variable specified by variable name vn. External procedure calls (ESCall) are stubs for a linker that eventually replaces them by ordinary procedure calls. Extended procedures (XCall) can return multiple values to a list of left-expressions. This is the only way for an XCall to manipulate the ordinary

Table 2 C0 Statements s

Skip	the empty statement
Comp s_1 s_2	sequential composition
Assign e_l e *sid*	assignment of expression e to left-expression e_l
PAlloc e_l *tn sid*	allocation of object of type name *tn* and assignment to pointer e_l
SCall *vn pn es sid*	procedure call of *pn* with parameters *es* and result *vn*
Return e *sid*	return from procedure
Ifte e s_1 s_2 *sid*	if-then-else with condition e
Loop e s *sid*	while loop with condition e and body s
Asm *ls sid*	inline assembly with instruction list *ls*
ESCall *vn pn es sid*	external procedure call
XCall *pn es* es_r *sid*	extended procedure call of *pn* with parameters *es* and result left-expressions es_r

C0 state, since it only operates on the extended state directly. Multiple return values nevertheless allow us to model updates to global variables or heap updates via pointers.

The extended state component on which XCalls operate is the same for all semantic layers. An extended procedure is defined semantically as a function that takes a list of parameter values and an extended state component and returns the new extended state component and a list of result values. This definition is then consulted to perform an XCall. Evaluating parameter expressions and returning the result values is handled by the C0 semantics itself. It depends on the semantical layer how the parameter and result values are represented. The small-step semantics flattens aggregate values to a list of bytes, whereas the big-step semantics and Hoare logic keep them together. Nevertheless, the definitions of the extended procedure definitions on the different layers have a lot in common since the core part is the transition on the extended state component. This makes the transfer of those definitions between the layers straightforward.

The different C0 semantics do not handle all statements. Neither of the C0 semantics handle assembly statements Asm, which are only meaningful in the assembly machines. We deal with C0 programs with inline assembly code by combining the C0 small-step semantics and the assembly semantics, both of which are tied together via compiler correctness. The small-step semantics describes the computation of the C0 program up to the point where an assembly statement is reached. The compiler correctness theorem allows us to relate the C0 small-step configuration to the assembly machine, which can then continue computation of the assembly parts. If the assembly part does not destroy any of the C0 invariants, the final assembly configuration can be mapped back to a C0 configuration from which the C0 computation can continue.

Furthermore, the Hoare logic and the big-step semantics do not support AddrOf expressions. This simplifies the memory model for global and local variables.

Regarding initialization of memory, C0 is quite strict. Heap memory is initialized by the (implementation of the) PAlloc statement. Global variables are initialized by the compiler. For local variables we employ a *definite assignment* analysis, which statically ensures that we only read from variables that were previously assigned to.

Definition 1 (Definite assignment analysis) The analysis has two parts:

$\mathcal{A}\ s$: is the set of variables that are guaranteed to be assigned to by any execution of statement s.

$\mathcal{D}\ s\ L\ A$: means that statement s passes the definite assignment check, with respect to the set of local variables L and the initially assigned variables A.

The analysis $\mathcal{A}\ s$ collects assignments to local as well as global variables. That is why this set is often intersected with the local variables L.

The typing constraints on C0 programs are not only employed for static typing, but are also used as typing invariants during execution of a statement. Hence they do not solely depend on static information but also on the dynamic configuration, e.g., to determine the type of a heap location. Since configurations are different for the big-step and the small-step semantics we postpone the definitions until Sections 3.2 and 3.3.

The core differences of the semantical layers of C0, which we introduce in the following sections, can be summarized as follows:

Hoare: split heap, aggregate values, implicit typing
Big-step: single monolithic heap, aggregate values, explicit typing
Small-step: single monolithic heap, flat values, explicit typing

These design decisions reflect the purpose of the layers. The Hoare logic is tuned to support verification of individual programs, whereas the small-step semantics is better suited for arguments about interleaving programs at the system level. C0 supports aggregate values, i.e., structures and arrays. As in hardware those values are broken down to a sequence of bytes in the small-step semantics. This allows us to calculate addresses of subcomponents in memory. At the big-step and the Hoare-level however, those aggregate values are stored in one 'memory cell'. One can still assign to subcomponents via ordinary left-expressions, but one cannot calculate the address of a subcomponent. The 'address-of' operator is not supported at the big-step and Hoare level. This rules out aliasing between different types and structure components. We exploit this guarantee by using a split heap memory model in the Hoare logic. Every component of a structure is stored in a separate heap [Bur72]. An assignment to one component does not affect any other components (of the same structure or other structures), which simplifies program verification. Type safety of C0 guarantees that this model can be mapped to the single monolithic heap of the other semantic layers.

Since we do not need a general theory of the C0 language at the Hoare logic level, we take the freedom to supply an individual state space for each program we verify. Every variable (and every split heap) becomes a field in a state space record, with its own HOL type. We employ the HOL type system to model C0 programming language types. Isabelle's type inference then takes care of typing constraints that would otherwise have to be explicitly maintained in the assertions. Again, type safety of C0 is the reason that allows us to connect the Hoare logic layer with the big-step semantics.

3.1 Hoare logic

The Hoare logic layer is somehow special compared to the big-step and small-step layer, since it is not C0 specific. It uses a general framework for the verification of sequential imperative programs within Isabelle/HOL [Sch05,Sch06]. The language model is called *Simpl*, and we embed C0 into it. The framework comes along with syntax, big- and small-step semantics, Hoare logics for partial as well as total correctness and an automated verification condition generator for Simpl. Soundness and completeness of the Hoare logic with respect to the operational semantics of Simpl is proven. We use the soundness theorem to transfer a Hoare triple about a C0 program to the operational semantics of Simpl. A correctness theorem about our embedding of C0 into Simpl then allows us to map these results to the big-step semantics of C0 [AHL^{+}08].

Since Simpl is generic it does not stipulate the representation of the state space, which is just a type variable. For our C0 instantiation we use a HOL record as state space, that we construct for a given program. The basic notions we need are big-step execution and guaranteed termination of Simpl programs.

Definition 2 (Big-step execution for Simpl) The judgment $\Gamma\vdash_{\mathsf{h}} \langle s,\ \sigma\rangle \Rightarrow \tau$ means that in context of program Γ the statement s started in initial state σ executes to the final state τ, where Γ maps procedure names to their bodies and σ and τ are states of the form $\textsc{Normal}\ \sigma'$ or $\textsc{Fault}\ f$, where f is an error flag.

Definition 3 (Guaranteed termination for Simpl) The judgment $\Gamma\vdash_{\mathsf{h}} s{\downarrow}\sigma$ means that in context of program Γ execution of statement s from initial state σ is guaranteed to terminate.

In this article we are concerned with total correctness properties of the form $\Gamma\models_{\mathsf{h}} P\ s\ Q$. Starting in a state that satisfies the precondition, execution of the statement is guaranteed to terminate without error and the final state satisfies the postcondition.

Definition 4 (Total correctness for Simpl)

$\Gamma\models_{\mathsf{h}} P\ s\ Q \equiv \forall\,\boldsymbol{\sigma}{\in}\textsc{Normal}\ `\ P.\ \Gamma\vdash_{\mathsf{h}} s{\downarrow}\boldsymbol{\sigma} \wedge (\forall\,\boldsymbol{\tau}.\ \Gamma\vdash_{\mathsf{h}} \langle s,\ \boldsymbol{\sigma}\rangle \Rightarrow \boldsymbol{\tau} \longrightarrow \boldsymbol{\tau} \in \textsc{Normal}\ `\ Q)$

3.2 Big-step semantics

A state σ in the big-step semantics is parametrized over the state extension $'x$ and is a record with the following components: the heap σ.heap maps locations to values, local variables σ.lcls and global variables σ.glbs map variable names to values, the natural number σ.$\mathsf{free\text{-}heap}$ indicates the available heap memory, and σ.ext is placeholder of type $'x$ for the extended state for XCalls.

A program Π is a four-tuple containing association lists for the various declarations of types, global variables, procedures, and extended procedures (for XCalls). We access this declaration information via the following functions: $\mathsf{tnenv}\ \Pi$ to resolve type names, $\mathsf{genv}\ \Pi$ to obtain type information for global variables, $\mathsf{plookup}\ \Pi$ and $\mathsf{xplookup}\ \Pi$ to lookup the definitions of procedures and XCalls.

Definition 5 (Big-step execution for C0) Judgment $\Pi{,}sz{,}L\vdash_{\mathsf{bs}} \langle s,\ \sigma\rangle \Rightarrow \tau$ means, that in context of program Π, the size function sz (to measure heap memory consumption of a type), and the set of local variables L, execution of the statement s in initial state σ leads to the final state τ.

The states σ and τ are of the form $\lfloor\sigma'\rfloor$ in case of normal execution or $\bot$ if an error occurred. The set of local variable names L is used to disambiguate accesses to local or global variables (local names hide global names). It is set to the local variables during a procedure call, whereas program Π and size function sz stay the same during the whole execution.

The size function sz is considered when allocating new heap memory. If enough memory is available a new object is created otherwise we nondeterministically return the null pointer or create a new object. We keep the size function as a parameter of the execution relation to gain some flexibility in when to discharge resource limitations. It shows up in variations also in the small-step semantics and in the compiler correctness theorem (cf. Sections 3.3 and 5.2) where it finally has to be discharged. If we choose to take a trivial size function (i.e., one that is constantly 0) we have to argue harder about memory restrictions later. Using a more appropriate size function allows us to carry out reasoning about memory restrictions already at the Hoare logic level (cf. Page 18).

Typing. The typing judgment for statements takes the static declaration information of the program into account. Moreover, as we also use typing to describe execution invariants it considers a heap typing HT, which maps locations to type names. In the big-step semantics the type information of heap locations is not maintained in the heap. This is not necessary for the execution of a C0 program since there is no runtime type information. For expressing type safety however we need this type information and introduce it via the heap typing.

Definition 6 (Typing of a statement) Judgment $xpt,pt,tt,VT,HT \vdash_{\mathsf{bs}} s \surd$ expresses that statement s is well-typed with respect to the extended procedure table xpt, procedure table pt, type table tt, variable typing VT, and the heap typing HT.

For a given program Π usually tt is obtained by $\mathsf{tnenv}\ \Pi$, pt by $\mathsf{plookup}\ \Pi$, and so forth. The variable typing VT is typically constructed by overwriting the global variable environment $\mathsf{genv}\ \Pi$ with the active local variable environment.

As a statement has to be well-typed the memories have to conform to the typing information. First we introduce a typing judgment for values, which we then extend to memories, by ensuring that every value is typed according to a memory typing MT.

Definition 7 (Typing of a value) The judgment $HT,tt \vdash_{\mathsf{bs}} v ::_{\mathsf{v}} T$ means that value v has type T in the context of heap typing HT and type table tt.

Definition 8 (Typing of memory)

$$HT,tt \vdash_{\mathsf{bs}} m :: MT \equiv \forall p\ v\ T.\ m\ p = \lfloor v \rfloor \wedge MT\ p = \lfloor T \rfloor \longrightarrow HT,tt \vdash_{\mathsf{bs}} v ::_{\mathsf{v}} T$$

Finally, we extend well-typedness to complete states. Local and global variables as well as the heap have to respect typing. For the heap, type information is obtained by composition of the type table tt and the heap typing HT. For global variables and the heap we additionally demand that at least as much locations or variables are initialized as in the typings. Global variables are always initialized and typing GT is constant during program execution. The heap typing HT gets extended as new objects are allocated. The number of allocated locations stays finite during execution. Local variables do not have to be initialized from the beginning of the procedure. However, the definite assignment analysis ensures that we only read initialized variables. In the invariants we use restricted local typings LT that correspond to the approximation of the definite assignment analysis.

Definition 9 (Typing of state)

$$\begin{aligned}&tt \vdash_{\mathsf{bs}} \sigma :: HT,LT,GT \equiv HT,tt \vdash_{\mathsf{bs}} \sigma.\mathsf{heap} :: (tt \circ_{\mathsf{m}} HT) \wedge \mathsf{dom}\ HT \subseteq \mathsf{dom}\ \sigma.\mathsf{heap} \wedge \\ &\quad \mathsf{finite}\ (\mathsf{dom}\ \sigma.\mathsf{heap}) \wedge HT,tt \vdash_{\mathsf{bs}} \sigma.\mathsf{lcls} :: LT \wedge HT,tt \vdash_{\mathsf{bs}} \sigma.\mathsf{glbs} :: GT \wedge \mathsf{dom}\ GT \subseteq \mathsf{dom}\ \sigma.\mathsf{glbs}\end{aligned}$$

Definition 10 (Valid programs) We define the predicate $\mathsf{valid\text{-}prog}\ \Pi$ to ensure basic well-formedness and typing properties for programs. In particular we require:

- identifiers within the different name spaces of type names, global variable names, local variable names, procedure names and extended procedure names are unique,
- every procedure body is well-typed and passes the definite assignment check, and
- the semantic definitions of the extended procedures respect their signature.

In contrast to a big-step semantics that only relates the initial to the final state of a statement execution, a small-step semantics describes single computation steps operating on configurations. A configuration c is a record with two components: the memory $c.\mathsf{mem}$ and the program rest $c.\mathsf{prog}$, which is a C0 statement. The memory configuration is a record with three components: the memory for global variables $c.\mathsf{mem}.\mathsf{gm}$, the heap memory $c.\mathsf{mem}.\mathsf{hm}$, and the frame stack $c.\mathsf{mem}.\mathsf{lm}$. Each frame is a pair containing the memory of local variables and the destination for the return value.

A memory m not only contains the values but also type and initialization information: the word addressable memory content $m.\mathsf{ct}$, a symbol table $m.\mathsf{st}$ associating variable names to types, and the set $m.\mathsf{init\text{-}vars}$ of initialized variables. Single memory cells of the small-step semantics are represented as an abstract data type and can store a single value of basic type. We have a constructor for every basic type, e.g., `unsigned` n for storing unsigned value n in a memory cell. In case of the heap memory the set of initialized variables and the variable names in the symbol table are meaningless, but the type information describes the objects in the heap.

We introduce a record for monolithic C0 small-step configurations c_{m} combining a small-step configuration $c_{\mathsf{m}}.\mathsf{conf}$, a type table $c_{\mathsf{m}}.\mathsf{tt}$ and a procedure table $c_{\mathsf{m}}.\mathsf{pt}$.

We refer to the memory objects of a C0 small-step configuration by the term generalized variables (or short: g-variable). This also includes all heap objects. Formally, g-variables are represented as a data type, e.g., GVAR-HM i for the i-th heap variable or GVAR-ARR g i for the i-th array element of g-variable g. Pointers are null (NULL) or point to a g-variable (PTR g).

Definition 11 (Transition function δ) The small-step transition function δ *tt pt enough-heap c* gets the type table *tt*, the procedure table *pt*, and the predicate *enough-heap* which takes the memory and a type as parameter and decides whether the allocation of a new object of that type succeeds. For property transfer between the big-step and the small-step semantics this has to be compatible with the *sz* function used there. If no fault occurs the result of the transition from the old configuration c is of the form $\lfloor c' \rfloor$, otherwise it is the error configuration $\perp$. We execute n steps by δ^n.

The transition function δ describes the C0 source level semantics of the compiler correctness theorem. It does not properly handle XCalls but returns a final configuration (where the program rest is SKIP). Hence, we extend it to handle XCalls:

Definition 12 (Transition function δ_{x}) The transition δ_{x} *tt pt enough-heap c x xpt* extends δ to handle XCalls on state extension x, according to the extended procedure table *xpt*. We execute n steps by δ_{x}^n.

A final configuration is reached when the program rest is SKIP. In this case the transition function is the identity. We define a relational view of the transition function that really stops in this configuration by a single introduction rule:

Definition 13 (Transition relation)

$$\frac{c.\mathsf{prog} \neq \textsc{Skip}}{tt,pt,enough\text{-}heap,xpt \vdash_{\mathsf{ss}} \lfloor (c,\, x) \rfloor \rightarrow \delta_{\mathsf{x}}\ tt\ pt\ enough\text{-}heap\ c\ x\ xpt}$$

We refer to the reflexive transitive closure by substituting the arrow $\rightarrow$ by $\rightarrow^*$.

As the symbol tables of the memories store typing information we do not need further components to describe well-typed configurations.

Definition 14 (Valid configurations) A configuration c is in set $\mathsf{valid\text{-}C0SS}$ tt pt $valid_{\mathsf{asm}}$ xpt if basic well-formedness and typing constraints hold, in particular:

- unique identifiers (cf. Definition 10),
- procedure bodies are well-typed and contain a single return statement at the end,
- the predicate $valid_{\mathsf{asm}}$ holds for all instructions in inline assembly statements, i.e., these instructions are well-typed,
- all memory frames are well-typed and contain only valid pointers, i.e., pointers which point to existing g-variables,
- the program rest conforms with the procedure table, i.e., all statements in the program rest are from one of the procedures and their order follows certain rules (cf. [Lei08, Section 5.4]), and
- the number of return statements in the program rest is strictly smaller than the number of stack frames.

3.4 Property transfer

Because the level of detail increases towards the lower layers, typically the amount of invariants on the configurations increases. For example take procedure calls. In the small-step semantics we explicitly maintain a frame stack. In a valid configuration there are at least as many frames on the stack as there are open return statements in the program rest. Already in the big-step semantics there is no need for a frame stack, because it abstracts the whole procedure call to one single step in the execution. In the light of property transfer the invariants of lower layers can be divided into two categories: (i) constraints necessary to establish the abstraction and to justify the property transfer and (ii) additional invariants only meaningful for the lower layers.

When composing results at the lower layer we have to ensure that the first category of invariants hold as a prerequisite to apply the transfer theorem. Moreover, the transferred result must maintain the second kind of invariants since those properties may be needed for further reasoning at the lower layer. For example consider reasoning about a procedure call on the small-step semantics. We are in a valid configuration with a certain frame stack and attempt to use a functional property of the procedure that we have proven using Hoare logics. Since the transfer theorem only works for well-typed configurations we have to know that in our current configuration the program is well-typed and that the memory also respects this typing. If also the precondition of the Hoare triple holds we can apply the property transfer theorem and derive that the postcondition holds for the final state of the procedure call. The postcondition itself only refers to the topmost frame, since the frame stack is no longer visible in the Hoare logic. Additionally, we want to ensure that the final configuration is still well-typed (type safety) and that the procedure call has only affected the topmost frame, so that properties on the lower frames still hold. These additional invariants appear in the following transfer theorems to preserve state information used for seamless integration and reasoning at the lower level.

The granularity on which we transfer Hoare triples down to the small-step semantics is procedure calls. As the transfer theorems employ constraints on the procedures, we have to be careful not to become too restrictive. Especially the step from the Hoare logic layer to the big-step semantics is critical since we switch from a shallow to a deep embedding of the state space. It would make no sense to define abstractions and prove properties about procedures, variables and types that are not subject of the transfer.

The transfer theorems are defined relative to the *program context* which consists of the type table, the procedure table, and the symbol table for global variables. At the Hoare layer we start with a minimal program context that is necessary to define the procedure we attempt to transfer. After transferring the result to the big-step layer we extend this minimal context to the target context. This way constraints on procedures only have to hold for the minimal context.

3.4.1 Hoare to big-step

The transfer of results from the Hoare logic level to the big-step level has one peculiarity: we switch from a shallow embedding of the state space to a deep embedding. At the big-step level all programming language values are embedded into a single data type and memory is formulated as a partial function from variable names to those values. This uniform representation is quite natural for the formulation of the language semantics. For the verification of individual programs within the Hoare logic however, every variable becomes an own field in the state space record, with its own HOL type. This unification of programming language typing with HOL typing brings the crucial benefit of automatic type inference to the Hoare logic, which relieves us from maintaining explicit typing constraints in the pre- and postconditions of Hoare triples. However, it comes at a cost that shows up when transferring results to the big-step level. As the state representation depends on the individual program we cannot generically define an abstraction function from a big-step memory configuration to a Hoare logic memory configuration. However, we can still develop a general property transfer theory by employing Isabelle's locales. We introduce abstraction functions for the core operations (lookup and update of atomic state components) as locale parameters, for which we *assume* commutativity properties. On the basis of these functions we can define further abstractions that allow the simulation of expression evaluation, left-expressions, assignments, and finally statements. This gives us an abstract theory of property transfer based on the assumptions of the locale. For any concrete program, where the state space in the Hoare logic is fixed, we can actually define the basic abstraction functions and discharge the assumptions of the locale by proving the commutativity properties. As the definition of these functions and the proofs are schematic we have automated the proofs with tactics.

In the course of building derived abstractions and proving properties on them more and more information about the big-step configuration has to be made available, in particular variable declarations with their type information. This information is made available via the following locale parameters: the type table tt, typing for global variables GT, local variables LT, and heap locations HT, and finally the C0 program Π, declared at the more detailed big-step level. The requirements on the parameters are straightforward (e.g., $tt = \mathsf{tnenv}\ \Pi$) and the intuition about them is sufficient for the course of this article. A thorough treatment can be found in [Sch06].

Both the abstraction function abs_{s} for a C0 statement and the abstraction function abs_{σ} for a big-step state are parametrized by a procedure name pn. Thereby the abstraction functions can discriminate between local and global variables. Moreover, a big-step state can correspond to several states in the Hoare logic. The reason is that the Hoare logic does not distinguish global and local variables, they lie side by side in the same state record. A big-step state only constrains the value of the active local variables, the variables of other procedures are irrelevant and hence any valuation of them in the Hoare logic is fine. Thus the function abs_{σ} yields a set of abstract states.

The main constraints we put on the program and the big-step configurations for which we attempt to transfer Hoare triples are type constraints. Only in a well-typed C0 setting the abstraction works out, since at the level of the Hoare logic typing is already enforced by Isabelle's type inference, due to the shallow embedding.

The cornerstone of the following transfer theorem is to strengthen the precondition from the abstract P_{h} to the big-step variant P and similarly to weaken the postcondition from the abstract Q_{h} to Q. These basic steps are decorated with validity constraints and frame conditions. The Hoare triple we attempt to transfer depends on the universally quantified auxiliary variable Z. By this the pre- and postcondition can be connected with each other [Kle99]. Typically Z fixes (parts of) the pre-state such that the post state can refer to it, e.g., to express frame conditions. Note that in the last premise of the theorem the auxiliary variable Z is existentially quantified and can depend on both versions of the initial state σ_{h} and σ, respectively.

The program has to be valid and the set of local variables L is determined by the current procedure pn. Most of following preconditions can rely on a big-step state σ that satisfies the precondition. For those constraints that directly refer to σ the reason is obvious. For the typing constraint on the statement s the dependency on σ is introduced via the heap typing HT. Imagine a pointer value as parameter of a procedure call specification. From the type constraints of the procedure we can infer the type name the pointer points to, which has to coincide with the heap typing HT at the location of the pointer. This location is part of the heap which also has to conform to the heap typing HT. For definite assignment the locations we assume to be assigned must be subset of the actually assigned values in the initial state. As the statement s we transfer is usually a procedure call, the definite assignment check can already be passed by an empty set A. This makes the subsumption test $A \subseteq \mathsf{dom}\ \sigma.\mathsf{lcls}$ trivial and also imposes no constraints on the typing of local variables since $LT\ pn{\restriction}_A$ becomes the empty map for $A = \{\}$. The way we construct the state abstraction ensures $\forall\, \sigma.\ abs_\sigma\ pn\ \sigma \neq \{\}$. The last premise is the promised connections between the preconditions P and P_{h} and the postconditions Q and Q_{h}. During execution of the statement s the heap may grow due to new memory allocations. Hence the heap typing for the final state HT' is an extension of the initial heap typing. Similar the statement may assign local variables which are predicted by $\mathcal{A}\ s$, and hence the final state must respect the typing of those additionally initialized variables. The last two conjuncts of the precondition in the final implication are pure frame conditions. All global variables that are not mentioned in the global typing GT and all heap locations that are not typed according to heap typing HT and the type table tt stay the same. These frame conditions are useful when we extend the typing environments, e.g., to embed the result in a bigger program.

Theorem 1 (Property transfer from Hoare to big-step)

$$
\begin{array}{l}
[\![\forall\, Z.\ \Gamma \models_{\mathsf{h}} (P_{\mathsf{h}}\ Z)\ abs_{\mathsf{s}}\ pn\ s\ (Q_{\mathsf{h}}\ Z);\ \mathsf{valid\text{-}prog}\ \Pi;\ L = \mathsf{dom}\ (LT\ pn);\\
\ \forall\, \sigma{\in}P.\ \mathsf{xplookup}\ \Pi{,}\mathsf{plookup}\ \Pi{,}tt{,}GT \mathbin{++} LT\ pn{,}HT \vdash_{\mathsf{bs}} s\ \surd;\ \mathcal{D}\ s\ L\ A;\ \forall\, \sigma{\in}P.\ A \subseteq \mathsf{dom}\ \sigma.\mathsf{lcls};\\
\ \forall\, \sigma{\in}P.\ tt \vdash_{\mathsf{bs}} \sigma{::}HT{,}LT\ pn{\restriction}_A{,}GT;\ \forall\, \sigma{\in}P.\ abs_\sigma\ pn\ \sigma \neq \{\};\\
\ \forall\, \sigma{\in}P.\ \forall\, \sigma_{\mathsf{h}} {\in} abs_\sigma\ pn\ \sigma.\\
\qquad\qquad \exists\, Z.\ \sigma_{\mathsf{h}} \in P_{\mathsf{h}}\ Z\ \wedge\\
\qquad\qquad\ \ (\forall\, HT'.\ HT \subseteq_{\mathsf{m}} HT' \longrightarrow\\
\qquad\qquad\qquad\quad (\forall\, \tau\ \tau_{\mathsf{h}}.\\
\qquad\qquad\qquad\qquad \tau_{\mathsf{h}} \in abs_\sigma\ pn\ \tau \wedge \tau_{\mathsf{h}} \in Q_{\mathsf{h}}\ Z \wedge tt \vdash_{\mathsf{bs}} \tau{::}HT'{,}LT\ pn{\restriction}_{(A\ \cup\ \mathcal{A}\ s)}{,}GT\ \wedge\\
\qquad\qquad\qquad\qquad A \cup L \cap \mathcal{A}\ s \subseteq \mathsf{dom}\ \tau.\mathsf{lcls}\ \wedge\\
\qquad\qquad\qquad\qquad (\forall\, n.\ n \notin \mathsf{dom}\ GT \longrightarrow \tau.\mathsf{glbs}\ n = \sigma.\mathsf{glbs}\ n)\ \wedge\\
\qquad\qquad\qquad\qquad (\forall\, l\ tn.\ HT\ l = \lfloor tn \rfloor \wedge tn \notin \mathsf{dom}\ tt \longrightarrow \tau.\mathsf{heap}\ l = \sigma.\mathsf{heap}\ l) \longrightarrow\\
\qquad\qquad\qquad\qquad \tau \in Q))]\!]\\
\Longrightarrow \Pi{,}sz{,}L \models_{\mathsf{bs}} P\ s\ Q
\end{array}
$$

This theorem is a corollary from simulation properties between the operational semantics of Simpl and C0, which are outlined in [AHL$^+$08] and detailed in [Sch06]. The core theorems are the correspondence of a C0 big-step execution and a Simpl execution and preservation of termination from Simpl to C0.[2] The simulation theorem is built from a couple of lemmas along the syntactic entities of C0: expression evaluation, left-expression evaluation, assignment, and statement execution. The latter one is proven by rule-induction on the big-step semantics, the former ones by induction on the (left-) expressions or the structure of the involved types. Most effort goes into the simulation of assignment. The reason for this are the differences in the representation of aggregate C0 values, which are broken down to their components in the split heap model of Simpl. Preservation of termination builds on the simulation theorem and is proven by induction on the termination relation. Altogether the verification effort sums up to about 0.75 person years. Some examples for applying the transfer theorems to simple procedures can be found in Section 8.8 of [Sch06].

3.4.2 Big-step to small-step

In addition to the standard constraints on configurations valid-C0SS we impose further restrictions for the purpose of property transfer. (i) Certain statements and expressions are not allowed since they are not supported by the big-step semantics or the Hoare logics, namely AddrOf, ESCall, and inline assembly. The function scalls collects the procedure calls of a statement and the set SCalls inductively collects further calls in the procedure environment for the given initial set. (ii) The global variables have to be initialized and we only allow pointers to (root) heap locations in memory (the small-step semantics additionally allows pointers to global variables or to subcomponents of an aggregate value).[3] (iii) Moreover, every type in the heap has to have a proper type name in the type environment. This is attributable to a subtle deviation in the different semantic formalizations. Whereas the small-step semantics directly maps pointers to types, the big-step semantics makes an indirection via type names. (iv) The small-step semantics maintains a frame stack for procedure calls where the destination for the return value is located (the second component of a frame). In the big-step semantics we directly store the return value within the local variables. For this purpose we reserve the variable name Res, which has to be distinct from other local and global variable names. Only root positions of local and global variables are valid return destinations. In case the return destination is a local variable it has to be defined in the frame stack below, formalized by the predicate valid-retvars. (v) Each procedure in the procedure environment has to pass the definite assignment check, where we assume parameters, local variables, and variable Res to be local, and the parameters to be initialized.[4] (vi) Similarly, the program rest has to pass the definite assignment check. However, as the program rest gets expanded during the small-step computation it may contain multiple returns and hence the program rest is split into several regions that correspond to procedure invocations on the frame stack. This generalization is formalized by $\mathcal{D}$s and LAs.

[2] Guaranteed termination form an initial configuration is formalized on top of the big-step semantics as an inductive definition.

[3] This can be liberated by only enforcing it for the pointers to known types, such that one can allow all kind of pointers to new types when embedding the result into a bigger context.

[4] We could liberate this by only enforcing it for the procedures in SCalls pt (scalls c.prog).

Definition 15 (Valid small-step configurations for property transfer)

$$
\begin{array}{l}
\mathsf{valid\text{-}cfg_{ss}}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ c\ x \equiv c \in \mathsf{valid\text{-}C0SS}\ tt\ pt\ valid_{\mathsf{asm}}\ xpt\ \wedge \\
\quad \mathsf{noAddrOf\text{-}Asm\text{-}ESCall}\ c.\mathsf{prog}\ \wedge \\
\quad (\forall\, p \in \mathsf{SCalls}\ pt\ (\mathsf{scalls}\ c.\mathsf{prog}).\ \mathsf{noAddrOf\text{-}Asm\text{-}ESCall}\ (\mathsf{the}\ (\mathsf{map\text{-}of}\ pt\ p)).\mathsf{proc\text{-}body})\ \wedge \\
\quad \mathsf{globals\text{-}initialized}\ c.\mathsf{mem.gm} \wedge \mathsf{only\text{-}heap\text{-}pointer}\ c.\mathsf{mem} \wedge \mathsf{named\text{-}types}\ tt\ (\mathsf{hm\text{-}st}\ c.\mathsf{mem})\ \wedge \\
\quad \mathsf{Res} \notin \mathsf{fst}\ `\ \mathsf{set}\ c.\mathsf{mem.gm.st} \wedge \mathsf{Res} \notin \mathsf{fst}\ `\ \bigcup \mathsf{set}\ `\ \mathsf{st}\ `\ \mathsf{fst}\ `\ \mathsf{set}\ c.\mathsf{mem.lm}\ \wedge \\
\quad (\forall\, gv \in \mathsf{snd}\ `\ \mathsf{set}\ c.\mathsf{mem.lm}.\ \mathsf{is\text{-}root\text{-}gvar}\ gv)\ \wedge \\
\quad \mathsf{valid\text{-}retvars}\ (\mathsf{snd}\ (\mathsf{hd}\ c.\mathsf{mem.lm}))\ (\mathsf{tl}\ c.\mathsf{mem.lm})\ \wedge \\
\quad (\forall\, p \in \mathsf{snd}\ `\ \mathsf{set}\ pt. \\
\qquad \mathtt{let}\ pns = \mathsf{map\ fst}\ p.\mathsf{proc\text{-}parameters};\ lns = \mathsf{map\ fst}\ p.\mathsf{proc\text{-}local\text{-}vars} \\
\qquad \mathtt{in}\ \mathcal{D}\ p.\mathsf{proc\text{-}body}\ (\mathsf{set}\ (pns\ @\ lns\ @\ [\mathsf{Res}]))\ (\mathsf{set}\ pns) \wedge \mathsf{Res} \notin \mathsf{set}\ pns \cup \mathsf{set}\ lns)\ \wedge \\
\quad \mathcal{D}\mathsf{s}\ c.\mathsf{prog}\ ((\mathsf{vnames}\ c,\ \mathsf{init\text{-}vnames}\ c) \cdot \mathsf{LAs}\ (\mathsf{snd}\ (\mathsf{hd}\ c.\mathsf{mem.lm}))\ (\mathsf{tl}\ c.\mathsf{mem.lm}))
\end{array}
$$

Our notion of a Hoare triple at the level of the small-step semantics is biased towards property transfer from the big-step level. We encode validity of configurations and transition invariants right into this notion and also carry some leftovers of the big-step configurations around, namely the set of local variables L and the program Π. The reason for this peculiarity is that the set of local variables and even the program declarations are directly encoded into each configuration in the small-step semantics. Hence we cannot relate those entities between the big- and the small-step semantics without referring to a small-step configuration which is only accessible within the pre- and postconditions of the Hoare triple.

We consider an initial configuration $(c,\ x)$. The memory of the configuration and the extended state fulfill the precondition. Moreover the initial configuration is valid, the program rest corresponds to statement s which does not contain a return statement. As we typically transfer a single procedure call this is trivially the case. The abstraction relations between the small-step configuration and the set of local variables L, as well as the program Π hold. Since we define total correctness non-terminating computations are ruled out, which is denoted by the precondition $\neg\ tt,pt,enough\text{-}heap,xpt \vdash_{\mathsf{ss}} \lfloor(c,\ x)\rfloor \rightarrow \ldots(\infty)$. For every computation the final configuration cx', must not be the error configuration, the postcondition has to hold, and it has to be valid. Moreover, the transition invariant between initial and final configuration has to be satisfied.

Definition 16 (Total correctness on small-step semantics)

$$
\begin{array}{l}
tt,pt,enough\text{-}heap,xpt,valid_{\mathsf{asm}},L,\Pi \models_{\mathsf{ss}} P\ s\ Q \equiv \\
\quad \forall\, c\ x.\ (c.\mathsf{mem},\ x) \in P \wedge \mathsf{valid\text{-}cfg_{ss}}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ c\ x \wedge c.\mathsf{prog} = s\ \wedge \\
\qquad\quad \mathsf{nr\text{-}returns}\ s = 0 \wedge L = \{\mathsf{Res}\} \cup \mathsf{fst}\ `\ \mathsf{set}\ (\mathsf{toplm\text{-}st}\ c.\mathsf{mem})\ \wedge \\
\qquad\quad \mathsf{absProg}\ tt\ pt\ (\mathsf{gm\text{-}st}\ c.\mathsf{mem})\ \Pi \longrightarrow \neg\ tt,pt,enough\text{-}heap,xpt \vdash_{\mathsf{ss}} \lfloor(c,\ x)\rfloor \rightarrow \ldots(\infty)\ \wedge \\
\qquad\quad (\forall\, cx'.\ tt,pt,enough\text{-}heap,xpt \vdash_{\mathsf{ss}} \lfloor(c,\ x)\rfloor \rightarrow^{*} cx' \wedge \mathsf{final}\ cx' \longrightarrow \\
\qquad\qquad (\exists\, c'\ x'\ xs.\ cx' = \lfloor(c',\ x')\rfloor \wedge (c'.\mathsf{mem},\ x') \in Q\ \wedge \\
\qquad\qquad\quad \mathsf{valid\text{-}cfg_{ss}}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ c'\ x'\ \wedge \\
\qquad\qquad\quad \mathsf{transition\text{-}invariant}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ s\ xs\ c\ x\ c'\ x'))
\end{array}
$$

The transition invariant captures essential invariants of the small-step computation that hold between the initial and final configuration of the procedure call that we transfer. First of all the computation only affects the topmost frame of local variables. Moreover, neither the type information of this frame nor of the global variables is changed. The type information for the heap memory may only grow. The second component of a frame stores the *left-value* of the return variable. This is also not modified by the current procedure call. Finally the increasing set of initialized local variables is approximated by the definite assignment analysis $\mathcal{A}$. In particular this ensures that the result variable of the procedure call is initialized in the final configuration.

Definition 17 (Transition invariant)

$\mathsf{transition\text{-}invariant}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ s\ xs\ c\ x\ c'\ x' \equiv \mathsf{tl}\ c'.\mathsf{mem}.\mathsf{lm} = \mathsf{tl}\ c.\mathsf{mem}.\mathsf{lm} \wedge$
$\quad \mathsf{toplm\text{-}st}\ c'.\mathsf{mem} = \mathsf{toplm\text{-}st}\ c.\mathsf{mem} \wedge \mathsf{gm\text{-}st}\ c'.\mathsf{mem} = \mathsf{gm\text{-}st}\ c.\mathsf{mem} \wedge$
$\quad \mathsf{hm\text{-}st}\ c'.\mathsf{mem} = \mathsf{hm\text{-}st}\ c.\mathsf{mem}\ @\ xs \wedge \mathsf{snd}\ (\mathsf{hd}\ c'.\mathsf{mem}.\mathsf{lm}) = \mathsf{snd}\ (\mathsf{hd}\ c.\mathsf{mem}.\mathsf{lm}) \wedge$
$\quad (\mathsf{fst}\ (\mathsf{hd}\ c.\mathsf{mem}.\mathsf{lm})).\mathsf{init\text{-}vars} \cup \mathsf{vnames}\ c \cap \mathcal{A}\ s \subseteq (\mathsf{fst}\ (\mathsf{hd}\ c'.\mathsf{mem}.\mathsf{lm})).\mathsf{init\text{-}vars}$

At its core the single premise of the following transfer theorem is a variant of an adaption theorem in Hoare logics. We strengthen the precondition and weaken the postcondition. Additionally, we take the different layers and the system invariants of the small-step layer into account. For an arbitrary initial configuration that fulfills the constraints of small-step validity and the precondition P we have to supply a big-step Hoare triple $\Pi,sz,L \models_{\mathsf{bs}} P_{\mathsf{bs}}\ s\ Q_{\mathsf{bs}}$ such that the big-step abstraction of the configuration fulfills the precondition P_{bs}. For a final valid small-step configuration, that fulfills the transition invariant and for which the big-step abstraction fulfills the postcondition Q_{bs} we have to derive the small-step postcondition Q. Note the existential quantification on the big-step pre- and postcondition. It is under the universal quantification of the initial configuration and hence can depend on this configuration. The function absState is used to abstract a small-step configuration to a big-step state.

Theorem 2 (Property transfer from big-step to small-step)

$\forall c\ x.\ \mathsf{valid\text{-}cfg}_{\mathsf{ss}}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ c\ x \wedge c.\mathsf{prog} = s \wedge \mathsf{nr\text{-}returns}\ s = 0 \wedge (c.\mathsf{mem},\ x) \in P \wedge$
$\qquad L = \{\mathsf{Res}\} \cup \mathsf{fst}\ `\ \mathsf{set}\ (\mathsf{toplm\text{-}st}\ c.\mathsf{mem}) \wedge \mathsf{absProg}\ tt\ pt\ (\mathsf{gm\text{-}st}\ c.\mathsf{mem})\ \Pi \longrightarrow$
$\qquad (\exists P_{\mathsf{bs}}\ Q_{\mathsf{bs}}.\ \Pi,sz,L \models_{\mathsf{bs}} P_{\mathsf{bs}}\ s\ Q_{\mathsf{bs}} \wedge \mathsf{absState}\ hs\ sz\ c.\mathsf{mem}\ x \in P_{\mathsf{bs}} \wedge$
$\qquad\quad (\forall c'\ x'\ xs.$
$\qquad\qquad \mathsf{valid\text{-}cfg}_{\mathsf{ss}}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ c'\ x' \wedge$
$\qquad\qquad \mathsf{transition\text{-}invariant}\ tt\ pt\ xpt\ valid_{\mathsf{asm}}\ s\ xs\ c\ x\ c'\ x' \wedge$
$\qquad\qquad \mathsf{absState}\ hs\ sz\ c'.\mathsf{mem}\ x' \in Q_{\mathsf{bs}} \longrightarrow (c'.\mathsf{mem},\ x') \in Q))$
$\Longrightarrow tt,pt,enough\text{-}heap,xpt,valid_{\mathsf{asm}},L,\Pi \models_{\mathsf{ss}} P\ s\ Q$

As for the transfer from Simpl to the C0 big-step semantics, the above theorem is a corollary from simulation properties between the big-step and the small-step semantics. To separate data-refinement arguments from computation steps, we introduce an intermediate small-step semantics [AHL$^+$08] that on the one hand shares the state space with the big-step semantics and on the other and has the same granularity of computation as the small-step semantics. We first prove that termination in the big-step semantics implies a terminating computation in the intermediate small-step semantics and that a terminating computation in the intermediate semantics has a corresponding execution in the big-step semantics. The second half is to prove that every computation step in the small-step semantics has a corresponding step in the intermediate semantics. Analogous to the simulation proofs behind Theorem 1 we build our lemmas along the syntactic entities of C0. As before the simulation of the assignment was the most time-consuming task, for similar reasons: the transition from aggregate values on the big-step level to flat values on the small-step level. The simulation between the big-step semantics and the intermediate semantics accounted to two weeks of work, whereas the second half from the intermediate semantics to the small-step semantics consumed almost one person year. The reason is both due to some technically involved arguments and intermediate notions, especially regarding the assignment, but also due to some minor deviations in the formalizations of corresponding aspects in the various semantics, which were developed at different sites. Some were adjusted and others were just bridged, because an accommodation of existing theories built on top of the semantics would have been too expensive.

Reasoning about heap consumption. In Theorem 2 we have hidden a precondition, which relates the enough heap predicate *enough-heap* and the size function *sz*:

$\neg$ *enough-heap* $m\ T \Longrightarrow$ free-heap-size $hs\ sz$ (hm-st m) $< sz\ T$

If according to *enough-heap* insufficient heap memory is available to allocate an object of type T in memory m, then this is consistent with the size function *sz*. The auxiliary function free-heap-size subtracts all the sizes of the type entries in the symbol table of the heap from the initial heap size *hs*. Note that we do not demand anything for the case that *enough-heap* is successful. The reason is that in the big-step semantics and the Hoare logic we treat memory allocation 'semi'-nondeterministically: if enough memory is left allocation always succeeds and returns a new pointer, otherwise it nondeterministically returns a new pointer or the null pointer. The motivation is the integration of garbage collection into the picture. In this scenario the Hoare logic only has an approximate knowledge about the free heap, since it just maintains a counter that is decremented by the size of the type for the freshly allocated object. The Hoare logic cannot compute heap consumption exactly because it only sees the current stack frame and, thus, does not know the set of reachable heap objects. So in case insufficient memory is predicted, this might not be the case in the implementation, since a garbage collector could have made more heap available in the meantime. This can be captured in a garbage collector aware *enough-heap* predicate.

Employing Hoare triples in redex position. At this point we are able to transfer a Hoare triple for a single procedure call to the small-step level. However, we typically attempt to integrate this procedure call into a bigger computation and hence the program rest in an intermediate configuration is not just this single procedure call, but rather the procedure call is at the redex position in the program rest. The following theorem allows us to employ the Hoare triple in such a configuration.

Theorem 3 (Employ Hoare triple in redex position)

$[\![tt,pt,\mathit{enough\text{-}heap},xpt,\mathit{valid}_{\mathsf{asm}},L,\Pi \models_{\mathsf{ss}} P\ s\ Q$; redex c.prog $= s$; (c.mem, x) $\in P$;
valid-cfg$_{\mathsf{ss}}{}'$ $tt\ pt\ xpt\ \mathit{valid}_{\mathsf{asm}}\ c\ x$; noAddrOf-Asm-ESCall (redex c.prog);
$\forall\, p \in$SCalls pt (scalls (redex c.prog)). noAddrOf-Asm-ESCall (the (map-of $pt\ p$)).proc-body;
nr-returns $s = 0$; $L = \{$Res$\} \cup$ fst ‘ set (toplm-st c.mem); absProg $tt\ pt$ (gm-st c.mem) $\Pi]\!]$
$\Longrightarrow \exists\, c'\ x'\ xs.\ tt,pt,\mathit{enough\text{-}heap},xpt \vdash_{\mathsf{ss}} \lfloor(c,\, x)\rfloor \rightarrow^* \lfloor(c',\, x')\rfloor\ \wedge$
    c'.prog = subst-redex SKIP c.prog $\wedge$ valid-cfg$_{\mathsf{ss}}{}'$ $tt\ pt\ xpt\ \mathit{valid}_{\mathsf{asm}}\ c'\ x'\ \wedge$
    transition-invariant $tt\ pt\ xpt\ \mathit{valid}_{\mathsf{asm}}\ s\ xs\ c\ x\ c'\ x' \wedge$ (c'.mem, x') $\in Q$

Note the subtle change in validity of a configuration. Whereas valid-cfg$_{\mathsf{ss}}$ demands that the whole program rest does not contain assembly, address-of or external calls (cf. Definition 15), valid-cfg$_{\mathsf{ss}}{}'$ drops this restriction. We ensure in the precondition that it holds for the redex though. One basic motivation of reasoning at the low abstraction level of the small-step semantics instead of the convenient Hoare logic level is the ability to combine ordinary C0-computation with inline assembly code. Hence, it would be worthless if we were incapable of using the transferred result in a situation where inline assembly code is part of the program rest. We start in a configuration where the redex of the program rest agrees with the statement in the Hoare triple. As we have total correctness we know that the computation leads to a state where the statement is completely executed, i.e., has evaluated to SKIP.[5] The program rest of

[5] Note that the small-step semantics is deterministic. Hence, restricting ourselves to one particular final configuration via the existential quantification is sufficient.

this configuration is obtained by substituting the redex position of the initial program rest with Skip. In the remainder of the theorem the usual properties about the initial and final configuration show up. The main argument in the proof is a straightforward consequence from the transition function extending the computation by unfolding the program rest in the redex position.

4 VAMP

The VAMP architecture is based on the DLX architecture [HP96] and was initially presented in [MP00]. An implementation of the VAMP has been formally verified in 2003 [BJK$^+$03, BJK$^+$06]. Since then, the VAMP has been extended with address translation and support for I/O devices [DHP05, AHK$^+$07, TS08].

There are three models related to the VAMP architecture; from concrete to abstract ones these are the VAMP's gate-level implementation, its instruction set architecture (ISA) specification, and its assembly language specification. For each model, one variant with device support and one variant without device support is defined. Simulation proofs relate two models of adjacent layers.

In this section we sketch the definitions of the models above the gate level and the relevant simulation results. Details on the gate-level implementation of the VAMP and its verification can be found elsewhere [TS08].

4.1 Assembly

The VAMP assembly language specification is intended to be a convenient layer for implementation and verification of low-level applications. Thus, it abstracts from certain aspects of lower layers which are irrelevant for most applications.

In the VAMP assembly machine, data is represented as integers while addresses are represented as naturals. This representation is optimized for applications working with integers; arguments regarding naturals and bit vector operations requires the use of conversion functions from / to integers. For example, the two functions to-nat32 and to-int32 convert between 32-bit integers and naturals.

A VAMP assembly configuration asm is a record with the following components: two program counters asm.dpc and asm.pcp for implementing the delayed branch mechanism, which hold the addresses of the current and next instruction, the general-purpose and special-purpose register files asm.gprs and asm.sprs, which are both lists of data, and the main memory asm.mm, which is a map from addresses to data. With $m[a,\ d]$ we obtain from memory m the content of length d starting at address a.

Definition 18 (Valid assembly configuration) A VAMP assembly configuration is called valid, denoted by the predicate valid-asm, if it fulfills certain basic well-formedness conditions: the program counters must be 32-bit naturals, register files must contain 32 registers, and all registers and memory cells must be 32-bit integers.

Instructions are represented with an abstract data type and converted between memory cells with conversion functions, e.g., with the function to-instr on instruction fetch. Thus, the function current-instr $asm \equiv$ to-instr (asm.mm (asm.dpc div 4)) denotes the instruction that will be executed next in the assembly machine. A VAMP assembly instruction is said to be valid if its register names and immediate constants are in the correct range. We denote this fact by the predicate $\mathsf{is\text{-}instr}$.

The VAMP assembly transition function δ_{asm} computes for a given assembly configuration *asm* the next configuration *asm*′. The transition is specified by a simple case distinction over $\mathsf{current\text{-}instr}$ *asm*. We use the notation $\delta^{n}_{\mathsf{asm}}$ *asm* to denote n steps of the assembly machine starting in configuration *asm*.

Definition 19 (Legal instructions) The assembly machine does not model execution of all instruction of the underlying ISA (cf. next section). Interrupt related instructions (RFE and TRAP) are handled by dummy transitions and we assume that they do not occur in the programs executed by the assembly machine (cf. Section 4.4). Access to special purpose registers is only supported in system mode. We formalize these restrictions with the predicate $\mathsf{legal\text{-}asm\text{-}instr}$ which is false for RFE and for instructions accessing a special purpose register in user mode.

4.2 Instruction set architecture

The VAMP instruction set architecture (ISA) serves as a specification of the VAMP gate-level hardware. There are three key differences of the VAMP ISA with respect to the more abstract VAMP assembly language: the VAMP ISA supports execution modes, address translation, and interrupt handling.

In *system mode*, programs can directly access the memory and fully control the architecture via a number of privileged instructions. In *user mode*, memory accesses are subject to address translation and attempts to execute a privileged instruction will result in an exception. The architecture responds to such exceptions, page-faults, and other interrupts by entering system mode and continuing execution at address 0, the start of the interrupt service routine (ISR). Normally, user mode is re-entered, continuing execution at the interrupted location, by issuing the privileged instruction RFE, which marks the end of the ISR.

Another, rather technical difference of the VAMP ISA is that some parts of its configuration and operation are defined in terms of bit vectors rather than naturals and integers, which is closer to the actual hardware implementation.

By design of the VAMP assembly model, no general simulation theorem can be established with respect to the VAMP ISA. However, equivalence can be established for system mode computations in which no interrupts occur, cf. Section 4.4. Similarly, when user mode computation is *virtualized* by the system software an equivalence result can also be established; this is an important part of the correctness of our CVM model discussed later in this article (cf. Section 6).

A configuration *isa* of the VAMP instruction set architecture is a record, which has the same components than a VAMP assembly configuration has, although differently typed:[6] the program counters *isa*.dpc and *isa*.pcp are bit vectors, the general-purpose and special-purpose registers *isa*.gprs and *isa*.sprs are functions from bit vectors to bit vectors, the main memory *isa*.mm is a function from bit vectors to pairs of bit vectors. Although bit vectors can have variable length in Isabelle, all bit vectors in the VAMP ISA have constant length, which is an invariant preserved during computation: register addresses are 5-bit bit vectors, register values are 32-bit bit vectors, which we also call *words*, and the main memory maps 29-bit double-word addresses to pairs of words.

[6] Older VAMP versions had floating-point registers and units [BJK+03], not needed here.

Transition function. The transition function of the VAMP instruction set architecture is denoted δ_{isa} *isa eev mifo*. It takes a current configuration *isa*, a bit vector of external interrupts (including a reset line), and a device output *mifo* as inputs. The latter two parameters mostly make sense if the VAMP is connected to devices (as sketched below). The transition function returns an updated configuration asm'.

Address translation. The VAMP provides a single-level address translation mechanism, which our CVM implementation uses to virtualize user process execution (cf. Sections 6 ff.). We sketch the relevant definitions of functions based on natural numbers, which is the form in which we will use them later.

VAMP's main memory is organized in pages, which are aligned chunks of data of size 2^{12} bytes. Two special purpose registers are relevant to address translation: The page table origin register PTO and the page table length register PTL designate a special region in main memory called *page table* consisting of word-sized page-table entries (PTEs). The page index $\mathsf{px}\ va \equiv va\ \mathsf{div}\ 2^{12}$ of a virtual address va is used as an index into this table. If $va > \mathsf{PTL}$ a page-fault exception is generated. Otherwise let *pte* denote the page table entry with index $\mathsf{px}\ va$. It consists of three components: (i) the physical-page index $ppx = pte\ \mathsf{div}\ 2^{12}$, (ii) the valid bit $v = pte\ \mathsf{div}\ 2^{11}\ \mathsf{mod}\ 2$, and (iii) the protection bit $p = pte\ \mathsf{div}\ 2^{10}\ \mathsf{mod}\ 2$. If the valid bit is on, read accesses to the virtual address are allowed. If additionally the protection bit is cleared, also writes are allowed. If these conditions are met, an access to a virtual address va will be performed on the physical address $ppx * 2^{12} + va\ \mathsf{mod}\ 2^{12}$.

4.3 Devices and their integration

Devices. We model devices as deterministic finite-state machines communicating with an external environment and the processor. The external environment is used to model non-determinism and communication; a network interface card, for example, sends and receives network packets. The processor accesses a device by reading or writing special addresses. The devices, in turn, can signal interrupts to the processor; Direct memory access (DMA) is not considered.

Formally, the interface between the processor and devices is defined by memory interface inputs *mifi* and outputs *mifo*, which encode the processor's request and the device's response to the request. A device of type x is represented by (i) types for its external input, external output, and configuration, (ii) a predicate $intr_{\mathsf{x}}$ on its configuration that indicates pending interrupts, and (iii) a transition function δ_{x}, which takes an input from the external environment, an input from the processor, and a device configuration as parameters and returns an updated configuration, an output to the processor, and an output to the external environment.

To generalize over different device types (hard disk, network card, etc.) we define a generalized external environment and device configurations as abstract data types. The transition function $\delta_{\mathsf{dev}}\ eifi\ mifi\ d = (d', mifo, eifo)$ can be used for multiple device types. Likewise, the predicate $\mathsf{intr}\ d$ indicates pending interrupts.

For the definition of non-interference predicates (and also for block operations, which are not needed here) we also need a transition functions $\delta^*_{\mathsf{dev}}\ eifis\ mifis\ d$ for many device steps, which consumes a list of memory and externals inputs.

Models with devices. At the gate-level hardware, up to eight devices operate in lock-step with the processor, running with the same clock. However, moving to the instruction set architecture and to assembly, we lose granularity and hence timing information. We compensate for this loss by introducing interleaved execution of devices and the processor. An oracle, called execution sequence, determines when some device or the processor takes a step.

Definition 20 (Execution sequence) An execution sequence element is equal to $\perp$ if at a certain point the processor takes a step or equal to $\lfloor(DID,\ \mathit{eifi})\rfloor$ if the device with identifier DID and external input eifi takes a step. An execution sequence $\mathit{seq}_{\mathsf{asm}}$ is a map from natural numbers to execution sequence elements.

First, we define the model VAMP assembly with devices.

Definition 21 (Configuration of VAMP assembly with devices) A configuration $\mathit{asm}_{\mathsf{D}}$ of the VAMP assembly with devices model is a record with two fields: $\mathit{asm}_{\mathsf{D}}.\mathsf{proc}$, the assembly state of the processor and $\mathit{asm}_{\mathsf{D}}.\mathsf{devs}$, a map from device identifiers to device configurations.

Definition 22 (Transitions of VAMP assembly with devices) Given the execution sequence $\mathit{seq}_{\mathsf{asm}}$, the current VAMP assembly with devices state $\mathit{asm}_{\mathsf{D}}$ and the number N, the transition $\Delta^{N}_{\mathsf{asm}}\ \mathit{seq}_{\mathsf{asm}}\ \mathit{asm}_{\mathsf{D}}$ returns the state reached after the execution of N steps of the sequence $\mathit{seq}_{\mathsf{asm}}$.

Second, we define the model VAMP ISA with devices.

Definition 23 (Configuration of VAMP ISA with devices) A configuration $\mathit{isa}_{\mathsf{D}}$ of the model VAMP ISA with devices is a record with two fields: $\mathit{isa}_{\mathsf{D}}.\mathsf{proc}$, the ISA state of the processor and $\mathit{isa}_{\mathsf{D}}.\mathsf{devs}$, a map from device identifiers to device configurations.

In the transition function of the VAMP ISA with devices the previously defined execution sequence is split into two separate functions, $\mathit{seq}_{\mathsf{isa}}$ and eifis. This is closer to the gate-level implementation and eases the correctness proof for the hardware. The first function determines for step N whether the processor or one of the devices is executing, the second function maps step numbers to inputs from the external environment. To convert both functions into a VAMP assembly execution sequence, we define the conversion function $\mathsf{to\text{-}seq}_{\mathsf{asm}}\ \mathit{seq}_{\mathsf{isa}}\ \mathit{eifis}\ N = \perp$ iff $\mathit{seq}_{\mathsf{isa}}\ N = \perp$, and $\mathsf{to\text{-}seq}_{\mathsf{asm}}\ \mathit{seq}_{\mathsf{isa}}\ \mathit{eifis}\ N = \lfloor(DID,\ \mathit{eifi})\rfloor$ otherwise where $\mathit{seq}_{\mathsf{isa}}\ N = \lfloor DID\rfloor$ and $\mathit{eifis}\ N = \mathit{eifi}$.

Definition 24 (Transitions of VAMP ISA with devices) Given a step number N, the execution sequence $\mathit{seq}_{\mathsf{isa}}$, the current state of VAMP ISA with devices $\mathit{isa}_{\mathsf{D}}$, and a sequence of external inputs eifis, the transition $\Delta^{N}_{\mathsf{isa}}\ \mathit{seq}_{\mathsf{isa}}\ \mathit{eifis}\ \mathit{isa}_{\mathsf{D}}$ returns the state reached after the execution of N steps.

4.4 Simulation theorems

As mentioned above the VAMP assembly model deliberately abstracts from modes and mode switches. Therefore, correctness of the VAMP assembly-ISA simulation could be described by two formal statements: for the system and user modes, respectively. In the following we present the simulation theorem only for the execution in the system mode, which will be used later on during kernel verification. For a user mode version, which requires virtualizing execution via address translation, see Section 6.

Abstraction relations. Both VAMP configurations (assembly and ISA) consist of the same components, but as their types are different we relate them by a type conversion.

Definition 25 (Assembly-ISA configuration equivalence) A VAMP assembly configuration *asm* and a VAMP ISA configuration *isa* are equivalent if (i) the contents of most registers in *asm* converted to bit vectors are equal to the corresponding values in *isa*,[7] and (ii) reading two consecutive cells in the assembly memory delivers the same pair as an ISA memory access:

$$\begin{array}{l}\mathsf{equiv\text{-}asm\text{-}isa}\ \mathit{asm}\ \mathit{isa} \equiv \mathit{isa}.\mathsf{dpc} = \mathsf{to\text{-}bv32}\ \mathit{asm}.\mathsf{dpc} \wedge \mathit{isa}.\mathsf{pcp} = \mathsf{to\text{-}bv32}\ \mathit{asm}.\mathsf{pcp} \wedge \\ \quad \mathsf{gprs\text{-}equiv}\ \mathit{asm}.\mathsf{gprs}\ \mathit{isa}.\mathsf{gprs} \wedge \mathsf{sprs\text{-}equiv}\ \mathit{asm}.\mathsf{sprs}\ \mathit{isa}.\mathsf{sprs} \wedge \\ \quad (\forall\, a{<}2^{32}.\ \mathit{isa}.\mathsf{mm}\ (\mathsf{to\text{-}bv29}\ (a\ \mathsf{div}\ 8)) = \mathsf{double\text{-}word\text{-}read}\ \mathit{asm}.\mathsf{mm}\ a)\end{array}$$

Definition 26 (Assembly-ISA program equivalence) Since one assembly instruction could be represented by many ISA instructions instr-equiv requires that the program in the VAMP ISA is obtained via an assembler:

$$\begin{array}{l}\mathsf{instr\text{-}equiv}\ \mathit{asm}\ \mathit{isa}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len} \equiv \\ \quad \mathit{isa}.\mathsf{mm}[4 * \mathit{start\text{-}addr}, \mathit{prog\text{-}len}] = \\ \quad \mathsf{instr\text{-}prog\text{-}to\text{-}VAMP\text{-}prog}\ (\mathsf{get\text{-}instr\text{-}list}\ \mathit{asm}.\mathsf{mm}\ (4 * \mathit{start\text{-}addr})\ \mathit{prog\text{-}len})\end{array}$$

Since no system code is written directly in the VAMP ISA language the code of our kernel program satisfies this requirement.

Definition 27 (Assembly-ISA abstraction relation) Combining VAMP configuration and program equivalence relations together we obtain the abstraction relation between VAMP assembly and VAMP ISA.

$$\begin{array}{l}\mathsf{equiv\text{-}asm\text{-}isa\text{-}with\text{-}instr}\ \mathit{asm}\ \mathit{isa}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len} \equiv \\ \quad \mathsf{equiv\text{-}asm\text{-}isa}\ \mathit{asm}\ \mathit{isa} \wedge \mathsf{instr\text{-}equiv}\ \mathit{asm}\ \mathit{isa}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len}\end{array}$$

In case both models have devices an additional relation for the latter is introduced: the equality in our case.

Definition 28 (Assembly-ISA abstraction relation with devices) The abstraction relation equiv-asm-isa-with-instr-dev extends the VAMP abstraction relation with an equivalence statement for devices:

$$\begin{array}{l}\mathsf{equiv\text{-}asm\text{-}isa\text{-}with\text{-}instr\text{-}dev}\ \mathit{asm}_{\mathsf{D}}\ \mathit{isa}_{\mathsf{D}}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len} \equiv \\ \quad \mathsf{equiv\text{-}asm\text{-}isa\text{-}with\text{-}instr}\ \mathit{asm}_{\mathsf{D}}.\mathsf{proc}\ \mathit{isa}_{\mathsf{D}}.\mathsf{proc}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len} \wedge \mathit{asm}_{\mathsf{D}}.\mathsf{devs} = \mathit{isa}_{\mathsf{D}}.\mathsf{devs}\end{array}$$

Next, we introduce necessary restrictions for the simulation.

Definition 29 (Preconditions for simulation) For an assembly configuration *asm* and a program of length *prog-len* stored starting at address *start-addr* the necessary preconditions for the simulation during n steps are given by the predicate precond-asm-isa which comprises the following facts: (i) the program is placed in non-device memory, (ii) the instructions of the program are valid, and (iii) for the whole computation the code region is not written to (i.e., there is no self-modification), all instruction and data accesses are aligned, the code does not jump outside the code region, the system mode is on and all external interrupts are masked, the instruction to be executed is neither RFE nor TRAP, and no devices are accessed. Formally:

[7] The contents of general-purpose register 0 (which is constantly zero for the VAMP) and some special-purpose registers are irrelevant.

$\mathsf{precond\text{-}asm\text{-}isa}\ asm\ start\text{-}addr\ prog\text{-}len\ n \equiv$
$\quad 4 * (start\text{-}addr + prog\text{-}len - 1) < \mathsf{devices\text{-}border} \wedge$
$\quad \mathsf{prog\text{-}properties}\ asm\ start\text{-}addr\ prog\text{-}len \wedge \mathsf{dynamic\text{-}properties}\ asm\ start\text{-}addr\ prog\text{-}len\ n$

where

$\mathsf{dynamic\text{-}properties}\ asm\ start\text{-}addr\ prog\text{-}len\ n \equiv$
$\quad \forall\, n'{<}n.\ \mathsf{step\text{-}properties}\ (\delta_{\mathsf{asm}}^{n'}\ asm)\ start\text{-}addr\ prog\text{-}len \wedge \mathsf{no\text{-}dev\text{-}touch\text{-}step}\ (\delta_{\mathsf{asm}}^{n'}\ asm)$

Note that in case some device is accessed the restrictions are defined by the predicate $\mathsf{precond\text{-}asm\text{-}isa\text{-}dev}\ asm\ start\text{-}addr\ prog\text{-}len\ seq_{\mathsf{asm}}\ N$ which differs from the one above in the term about devices; instead of $\mathsf{dynamic\text{-}properties}$ we use

$\mathsf{dynamic\text{-}properties\text{-}dev}\ asm_{\mathsf{D}}\ start\text{-}addr\ prog\text{-}len\ seq_{\mathsf{asm}}\ N \equiv$
$\quad \forall\, N'{<}N.\ \mathsf{step\text{-}properties}\ (\mathsf{fst}\ (\Delta_{\mathsf{asm}}^{N'}\ seq_{\mathsf{asm}}\ asm_{\mathsf{D}})).\mathsf{proc}\ start\text{-}addr\ prog\text{-}len \wedge$
$\quad\quad \mathsf{dev\text{-}touch\text{-}correct\text{-}step}\ (\mathsf{fst}\ (\Delta_{\mathsf{asm}}^{N'}\ seq_{\mathsf{asm}}\ asm_{\mathsf{D}})).\mathsf{proc}$

Here, the predicate $\mathsf{dev\text{-}touch\text{-}correct\text{-}step}$ ensures word access to devices.

Finally, we introduce validity assumptions over execution sequences and sequences of inputs from the external environment.

Definition 30 (Sequences assumption) The predicate $\mathsf{precond\text{-}seq\text{-}isa}$ states that (i) an execution sequence seq_{isa} is live with respect to the process and all devices, and (ii) each element of external inputs $eifis$ has the type suitable for the device which is supposed to make a step at the corresponding position of seq_{isa}:

$\mathsf{precond\text{-}seq\text{-}isa}\ seq_{\mathsf{isa}}\ eifis\ devs \equiv \mathsf{proc\text{-}live\text{-}input\text{-}seq\text{-}isa}\ seq_{\mathsf{isa}} \wedge$
$\quad \mathsf{dev\text{-}live\text{-}input\text{-}seq\text{-}isa}\ seq_{\mathsf{isa}} \wedge \mathsf{eifis\text{-}welltyped\text{-}isa}\ seq_{\mathsf{isa}}\ devs\ eifis$

Next, we present simulation theorems between VAMP assembly and VAMP ISA for the system mode. They come in two flavors – without and with devices access. Regardless of the case the target model of the simulation is VAMP ISA with devices. The idea behind the theorems is as follows. We start with an assembly and an ISA machine between which the appropriate abstraction relation holds. We execute an assembly program on the assembly machine (with devices, in the second case) and find the corresponding number of steps of the ISA machine with devices, such that the abstraction relation is preserved.

VAMP assembly-ISA simulation without device 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. Since the target architecture has devices the simulation theorem has to conclude not only equivalence between VAMP assembly and VAMP ISA configurations but also that devices perform steps triggered only by the external environment, i.e., steps with the idle inputs from the processor. Next, we define a predicate that compares two configurations of device systems and determines whether the second is obtained from the first by taking only external steps.

Definition 31 (Non-interference of devices and the processor) The predicate $\mathsf{untouched\text{-}devs}$ checks whether the configuration of a device system $devs'$ is obtained from the configuration $devs$ by executing independently all devices steps contained in the prefix of an execution sequence seq_{isa} of length N with the external inputs $eifis$:

$$\begin{array}{l}
\textsf{untouched-devs}\ \mathit{devs}\ \mathit{devs}'\ \mathit{eifis}\ \mathit{seq}_{\textsf{isa}}\ N \equiv \\
\quad \forall\, \mathit{did}.\ \mathit{devs}'\ \mathit{did} = \\
\qquad \textsf{fst}\ (\delta^{*}_{\textsf{dev}}\ (\textsf{map}\ \mathit{eifis}\ (\textsf{filter-devs-isa}\ \mathit{seq}_{\textsf{isa}}\ \mathit{did}\ N)) \\
\qquad\quad (\textsf{replicate}\ |\textsf{filter-devs-isa}\ \mathit{seq}_{\textsf{isa}}\ \mathit{did}\ N|\ \textsf{mifi-limit-idle})\ (\mathit{devs}\ \mathit{did}))
\end{array}$$

Theorem 4 (Assembly-ISA simulation without devices access) *Let asm be the configuration of an assembly machine which executes in* n *steps a program that occupies prog-len words of memory starting at the address start-addr. Let* isa *and* devs *be the configuration of an ISA machine with devices, let* $\mathsf{seq}_{\mathsf{isa}}$ *be its execution sequence, and let* eifis *be the sequence of external inputs. Provided that (i) the assembly machine configuration is valid and the sequences assumptions hold, (ii) the preconditions for simulation are fulfilled, and (iii) the assembly-ISA equivalence relation holds, there exists a resulting ISA configuration* isa' *with devices* devs' *reached in* N *steps, such that the assembly-ISA equivalence relation is preserved and the property of non-interference of devices and the processor holds:*

$$\begin{array}{l}
[\![\textsf{valid-asm}\ \mathit{asm};\ \textsf{precond-asm-isa}\ \mathit{asm}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len}\ n; \\
\ \textsf{equiv-asm-isa-with-instr}\ \mathit{asm}\ \mathit{isa}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len};\ \textsf{precond-seq-isa}\ \mathit{seq}_{\textsf{isa}}\ \mathit{eifis}\ \mathit{devs}]\!] \\
\Longrightarrow \exists\, N\ \mathit{isa}'\ \mathit{devs}'. \\
\qquad \textsf{fst}\ (\Delta^{N}_{\textsf{isa}}\ \mathit{seq}_{\textsf{isa}}\ \mathit{eifis}\ (\!|\textsf{proc} = \mathit{isa},\ \textsf{devs} = \mathit{devs}|\!)) = (\!|\textsf{proc} = \mathit{isa}',\ \textsf{devs} = \mathit{devs}'|\!) \wedge \\
\qquad \textsf{equiv-asm-isa-with-instr}\ (\delta^{n}_{\textsf{asm}}\ \mathit{asm})\ \mathit{isa}'\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len} \wedge \\
\qquad \textsf{untouched-devs}\ \mathit{devs}\ \mathit{devs}'\ \mathit{eifis}\ \mathit{seq}_{\textsf{isa}}\ N
\end{array}$$

VAMP assembly-ISA simulation with devices access. The theorem about simulation of an assembly with devices by an ISA with devices is applied while reasoning about correctness of assembly programs which communicate with devices. In case devices are present on both levels the theorem becomes easier since there is a step-to-step correspondence between the models. Recall that the function $\textsf{to-seq}_{\textsf{asm}}\ \mathit{seq}_{\textsf{isa}}\ \mathit{eifis}$ converts an execution sequence and a list of externals for the VAMP ISA with devices model to an execution sequence for the VAMP assembly with devices model.

Theorem 5 (Assembly-ISA simulation with devices access) *Let* $\mathit{asm}_{\mathsf{D}}$ *be the configuration of an assembly machine with devices which executes in N steps a program that occupies prog-len words of memory starting at the address start-addr. Let* $\mathsf{isa}_{\mathsf{D}}$ *be the configuration of an ISA machine with devices, let* $\mathsf{seq}_{\mathsf{isa}}$ *be its execution sequence, and let* eifis *be the sequence of external inputs. Under the same assumptions as in Theorem 4 the assembly-ISA abstraction relation with devices holds at each step:*

$$\begin{array}{l}
[\![\textsf{valid-asm}\ \mathit{asm}_{\textsf{D}}.\textsf{proc}; \\
\ \textsf{precond-asm-isa-dev}\ \mathit{asm}_{\textsf{D}}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len}\ (\textsf{to-seq}_{\textsf{asm}}\ \mathit{seq}_{\textsf{isa}}\ \mathit{eifis})\ N; \\
\ \textsf{equiv-asm-isa-with-instr-dev}\ \mathit{asm}_{\textsf{D}}\ \mathit{isa}_{\textsf{D}}\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len}; \\
\ \textsf{precond-seq-isa}\ \mathit{seq}_{\textsf{isa}}\ \mathit{eifis}\ \mathit{isa}_{\textsf{D}}.\textsf{devs};\ N' \le N]\!] \\
\Longrightarrow \textsf{equiv-asm-isa-with-instr-dev}\ (\textsf{fst}\ (\Delta^{N'}_{\textsf{asm}}\ (\textsf{to-seq}_{\textsf{asm}}\ \mathit{seq}_{\textsf{isa}}\ \mathit{eifis})\ \mathit{asm}_{\textsf{D}})) \\
\qquad (\textsf{fst}\ (\Delta^{N'}_{\textsf{isa}}\ \mathit{seq}_{\textsf{isa}}\ \mathit{eifis}\ \mathit{isa}_{\textsf{D}}))\ \mathit{start\text{-}addr}\ \mathit{prog\text{-}len}
\end{array}$$

5 Compiling C0 to VAMP

Most software in Verisoft has been implemented and verified at the C0 level. Such C0 programs need to be translated (via assembly code) to machine code to be executable on the target machine. We have developed and verified a simple non-optimizing compiler from C0 to the assembly language of the VAMP processor [LP08].

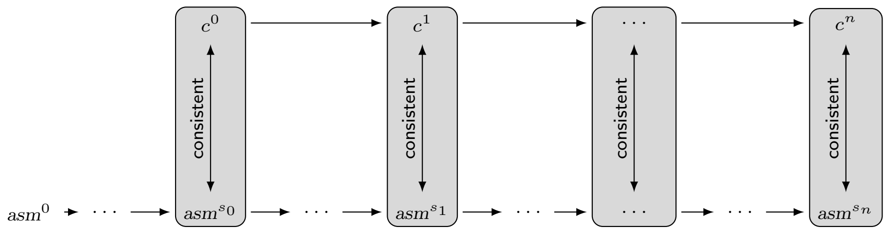


Fig. 3 Small-step compiler simulation theorem

Two versions of the compiler have been developed: a *compiling specification* in the functional specification language of Isabelle/HOL and a *compiler implementation* in C0. Accordingly, the compiler verification is split into two parts. First, we have shown that for a given C0 program the compiler implementation generates the same assembly code as specified by the compiling specification [Pet07]. Second, we have proven a simulation theorem between the original C0 program executed by the C0 small-step semantics and the generated assembly code executed by the VAMP assembly machine [Lei08].

To enable property transfer from the C0 to the VAMP layer, the simulation theorem has to meet special requirements. In particular, the simulation theorem is formulated based on small-step semantics to allow for reasoning about non-terminating and interleaving programs. Additionally, the compiler correctness proof lifts resource restrictions from the hardware layer to the more abstract C0 layer.

Here, we will briefly describe the induction step of the compiler simulation theorem. Details that have been omitted here can be found in [Lei08].

5.1 Abstraction relation: C0 to VAMP assembly

In essence, the compiler simulation theorem (cf. Figure 3) shows that every step i of the source program executed by the C0 small-step semantics is simulated by a certain number s_i of steps of the VAMP assembly machine executing the compiled code.

Definition 32 (Abstraction relation) The abstraction relation consistent states that a VAMP assembly configuration *asm* encodes a C0 configuration c via some allocation function *alloc* which maps C0 g-variables to their allocated address in the assembly machine. This relation is a conjunction of control, code, and data consistency relations defined later:

consistent $tt\ pt\ c\ alloc\ asm \equiv$ c-consistent $tt\ pt\ c\ asm \wedge$ code-consistent $tt\ pt\ c\ asm \wedge$
d-consistent $tt\ pt\ c\ alloc\ asm$

Definition 33 (Decomposition of a statement) With $\lceil s \rceil$ we obtain a list of sub-statements of s by decomposing the outer layer of sequential composition (loop bodies and if-then-else branches are not decomposed). Additionally, $\lceil\!\lceil x \rceil\!\rceil = [s {\in} \lceil x \rceil\ .\ s \neq \textsc{Skip}]$ removes any top-level Skip statements from the statement list.

Definition 34 (Control consistency) Control consistency states that the program counters of the VAMP correctly follow the control flow of the source program, i.e., that they point to the code of the first statement in the current program rest and that the return addresses stored in the run-time stack correspond with the structure of the program rest (this is hidden in the predicate ra-consistent):

$$\begin{aligned}
&\mathsf{c\text{-}consistent}\ tt\ pt\ c\ asm \equiv\\
&\quad [\![c.\mathsf{prog}]\!] \neq [] \longrightarrow asm.\mathsf{dpc} = \mathsf{the}\ (\mathsf{code\text{-}base}\ tt\ (\mathsf{gm\text{-}st}\ c.\mathsf{mem})\ pt\ (\mathsf{hd}\ [\![c.\mathsf{prog}]\!]))\ \wedge\\
&\quad asm.\mathsf{pcp} = asm.\mathsf{dpc} + 4 \wedge \mathsf{ra\text{-}consistent}\ tt\ pt\ c\ asm
\end{aligned}$$

Definition 35 (Code consistency) Code consistency requires that the compiled code of the C0 program is stored at address program-base of the VAMP machine. For technical reasons, we also require that all bit patterns stored in the code region can be decoded by the VAMP processor.

$$\begin{aligned}
&\mathsf{code\text{-}consistent}\ tt\ pt\ c\ asm \equiv\\
&\quad \mathtt{let}\ code = \mathsf{codegen\text{-}program}\ tt\ pt\ (\mathsf{gm\text{-}st}\ c.\mathsf{mem})\\
&\quad \mathtt{in}\ \mathsf{get\text{-}instr\text{-}list}\ asm.\mathsf{mm}\ \mathsf{program\text{-}base}\ |code| = code\ \wedge\\
&\qquad (\forall x \in \mathsf{set}\ (asm.\mathsf{mm}[\mathsf{program\text{-}base},\ |code|]).\ \mathsf{decodable}\ x)
\end{aligned}$$

Definition 36 (Data consistency) Data consistency ensures that all reachable g-variables of the C0 program, the heap, and the run-time stack are properly represented in the assembly machine. It is formalized as a conjunction of several predicates.

$$\begin{aligned}
&\mathsf{d\text{-}consistent}\ tt\ pt\ c\ alloc\ asm \equiv \mathsf{v\text{-}consistent}\ tt\ c\ alloc\ asm \wedge \mathsf{p\text{-}consistent}\ tt\ c\ alloc\ asm\ \wedge\\
&\quad \mathsf{a\text{-}consistent}\ tt\ pt\ c\ alloc \wedge \mathsf{fh\text{-}consistent}\ tt\ pt\ c\ alloc\ asm \wedge \mathsf{r\text{-}consistent}\ tt\ pt\ c\ alloc\ asm
\end{aligned}$$

The two main predicates v-consistent and p-consistent ensure that all reachable g-variables of the C0 machine are properly represented in the assembly machine.

Definition 37 (Value consistency) Value consistency v-consistent requires for all initialized *reachable* g-variables g of basic type that C0 and VAMP machine store the same value.

$$\begin{aligned}
&\mathsf{v\text{-}consistent}\ tt\ c\ alloc\ asm \equiv\\
&\quad \forall g.\ g \in \mathsf{reachable\text{-}gvars}\ tt\ c.\mathsf{mem} \wedge \mathsf{elementary\text{-}gvar}\ tt\ (\mathsf{symbols}\ c.\mathsf{mem})\ g\ \wedge\\
&\qquad \neg\ \mathsf{pointer\text{-}gvar}\ tt\ (\mathsf{symbols}\ c.\mathsf{mem})\ g \wedge \mathsf{gvar\text{-}initialized}\ c.\mathsf{mem}\ g \longrightarrow\\
&\qquad \mathsf{values\text{-}match}\ ((\mathsf{get\text{-}data}\ tt\ c.\mathsf{mem}\ g)\ 0)\ (asm.\mathsf{mm}\ (\mathsf{fst}\ (alloc\ g)\ \mathsf{div}\ 4))
\end{aligned}$$

Definition 38 (Pointer consistency) Pointer consistency p-consistent requires for all reachable pointer g-variables p which point to some g-variable g that the value stored at the allocated address of p in the assembly machine is the allocated base address of g. This defines a subgraph isomorphism between the reachable portions of the heaps of the C0 machine and the assembly machine.

$$\begin{aligned}
&\mathsf{p\text{-}consistent}\ tt\ c\ alloc\ asm \equiv\\
&\quad \forall p.\ p \in \mathsf{reachable\text{-}gvars}\ tt\ c.\mathsf{mem} \wedge \mathsf{pointer\text{-}gvar}\ tt\ (\mathsf{symbols}\ c.\mathsf{mem})\ p\ \wedge\\
&\qquad \mathsf{gvar\text{-}initialized}\ c.\mathsf{mem}\ p \longrightarrow\\
&\qquad (\mathsf{case}\ \mathsf{to\text{-}ptr}\ ((\mathsf{get\text{-}data}\ tt\ c.\mathsf{mem}\ p)\ 0)\ \mathtt{of}\\
&\qquad\ \textsc{ptr}\ g \Rightarrow \mathsf{to\text{-}nat32}\ (asm.\mathsf{mm}\ (\mathsf{fst}\ (alloc\ p)\ \mathsf{div}\ 4)) = \mathsf{fst}\ (alloc\ g)\\
&\qquad\ |\ \textsc{Null} \Rightarrow asm.\mathsf{mm}\ (\mathsf{fst}\ (alloc\ p)\ \mathsf{div}\ 4) = 0)
\end{aligned}$$

The remaining predicates of data consistency check technical details. Allocation consistency a-consistent ensures that the allocation function *alloc* is well defined (e.g., that the allocated memory regions of reachable g-variables do not overlap). Frame header consistency fh-consistent ensures that the run-time stack is properly represented in the assembly machine. Register consistency r-consistent checks the values of three special registers which store the base address of the global memory frame, the base address of the current stack frame, and the first unused heap address.

We slightly extend the requirements on valid C0 configurations from Section 3.3 for the compiler simulation proof by strengthening the requirements regarding the relation between the number of returns in the program rest and the recursion depth of the C0 configuration. For technical reasons we require that the recursion depth equals the number of return statements plus one (this property is fulfilled automatically for valid C0 programs which are started in an initial configuration).

The compiled code simulates the original C0 program only if no resource restrictions of the target machine are violated. There are two kinds of resource restrictions.

The *static* resource restrictions require that the generated code fits into the memory of the target machine. They are formalized via the set translatable-programs. Our simple code generation algorithm further restricts the source program: the size of loop bodies and the distance between function calls and the target function are limited by the size of immediate constants in VAMP assembly instructions. The size of expressions is bounded because intermediate results are not stored in memory but kept in registers.

Definition 39 (Valid C0 configurations) We combine the extended requirements in the following predicate.

$\mathsf{valid\text{-}C0\text{-}conf}\ tt\ pt\ xpt\ c \equiv c \in \mathsf{valid\text{-}C0SS}\ tt\ pt\ \mathsf{valid}_{\mathsf{asm}}\ xpt\ \wedge$
$\quad \mathsf{nr\text{-}toplevel\text{-}returns}\ \lceil c.\mathsf{prog}\rceil + 1 = \mathsf{recursion\text{-}depth}\ c.\mathsf{mem}\ \wedge$
$\quad (tt,\ pt,\ \mathsf{gm\text{-}st}\ c.\mathsf{mem}) \in \mathsf{translatable\text{-}programs}$

The *dynamic* resource restrictions require that the memory areas for stack and heap do not overflow. In contrast to the static restrictions, the dynamic restrictions must be checked for every step of the C0 small-step semantics. According to the memory layout of the C0 compiler both stack and heap grow upwards from fixed start addresses. Thus, we have to check that this start address plus the current size of stack or heap is not larger than the heap start address or the maximum address, respectively.

Definition 40 (Sufficient memory) We formalize the dynamic resource restrictions in the following predicate.

$\mathsf{sufficient\text{-}memory}\ max\text{-}address\ tenv\ pt\ c \equiv$
$\quad \mathsf{abase\text{-}local\text{-}frame}\ tenv\ pt\ (\mathsf{symbols}\ c.\mathsf{mem})\ (\mathsf{recursion\text{-}depth}\ c.\mathsf{mem}) \leq \mathsf{heap\text{-}base}\ \wedge$
$\quad \mathsf{heap\text{-}base} + \mathsf{asize\text{-}heap}\ c.\mathsf{mem}.\mathsf{hm}.\mathsf{st} < max\text{-}address$

Additionally, this version of the C0 compiler does not deal with failing memory allocation.[8] The simulation proof assumes that allocation of new memory always succeeds, i.e., that the predicate *enough-heap* (cf. Section 3.3) never returns False when allocation is being done.

$\mathsf{allocation\text{-}succeeds}\ tt\ enough\text{-}heap\ c \equiv$
$\quad \forall\, e\ sid\ tn.\ \mathsf{hd}\ \lceil c.\mathsf{prog}\rceil = \textsc{PAlloc}\ e\ tn\ sid \longrightarrow enough\text{-}heap\ c.\mathsf{mem}\ (\mathsf{the}\ (\mathsf{map\text{-}of}\ tt\ tn))$

This requirement can be easily achieved by instantiating the predicate *enough-heap* by *enough-heap* = ($\lambda m\ t.$ True), which we have done in the remaining sections. The drawback of this solution is that we no longer obtain useful information about heap consumption from the Hoare logic layer. Instead, this property has to be proven manually via the above mentioned predicate sufficient-memory.

[8] The compiler version described in [Lei08] does deal with failing allocation; see Chapter 11 of that work for a list of differences between both versions.

Definition 41 (Successful execution of assembly code) We introduce the notation $crange, arange \vdash_{\mathsf{asm}} asm \rightarrow^n asm'$ which means that the VAMP assembly machine successfully executes from configuration asm to asm' in n steps while respecting address range $arange$ and code range $crange$; observe, that the latter forbids self-modifying code. Additionally, it states that no exceptions have been generated and that no illegal instruction has been executed. Observe, that these properties have to be shown for *all* intermediate steps of the assembly machine. Formally, we define

$$
\begin{array}{l}
crange, arange \vdash_{\mathsf{asm}} asm \rightarrow^n asm' \equiv asm'.\mathsf{pcp} = asm'.\mathsf{dpc} + 4 \wedge \delta^n_{\mathsf{asm}}\ asm = asm' \wedge \\
\quad \mathsf{valid\text{-}asm}\ asm' \wedge \\
\quad (\forall\, n' < n.\ \neg\ \mathsf{mem\text{-}write\text{-}inside\text{-}range}\ (\delta^{n'}_{\mathsf{asm}}\ asm)\ crange\ \wedge \\
\qquad \mathsf{mem\text{-}access\text{-}inside\text{-}range}\ (\delta^{n'}_{\mathsf{asm}}\ asm)\ arange \wedge \mathsf{inside\text{-}range}\ crange\ (\delta^{n'}_{\mathsf{asm}}\ asm).\mathsf{dpc} \wedge \\
\qquad \neg\ \mathsf{is\text{-}exception}\ (\delta^{n'}_{\mathsf{asm}}\ asm) \wedge \mathsf{legal\text{-}asm\text{-}instr}\ (\delta^{n'}_{\mathsf{asm}}\ asm)\ (\mathsf{current\text{-}instr}\ (\delta^{n'}_{\mathsf{asm}}\ asm)))
\end{array}
$$

Successful execution of assembly code depends on two preconditions: the initial assembly configuration asm needs to be valid, i.e., $\mathsf{valid\text{-}asm}\ asm$, and the program counters have to be aligned on word boundaries and must not start in a delay slot (i.e., we require $asm.\mathsf{pcp} = asm.\mathsf{dpc} + 4$).

Using the predicates from the previous sections we can now easily formulate the induction step lemma of the compiler simulation theorem. Observe, that we do not only show that the abstraction relation is preserved but show additional properties which are used for the CVM correctness proofs.

Theorem 6 (Compiler induction step) *Assume the requirements from the previous sections and additionally that the next step of the C0 small-step semantics does not fail, that the program rest does not start with an inline assembly statement (inline assembly code is handled separately [IT08, ST08a]), that the program rest is not empty, that no stack or heap overflow occurs in the successor configuration c', and that the maximum address is within the address space of the VAMP. Then we show that there exists a step number n, a new allocation function alloc', and a new assembly configuration asm' such that*

1. *the assembly machine advances in n steps from asm to asm' without generating an interrupt,*
2. *the new assembly configuration asm' encodes the configuration c' via allocation function alloc',*
3. *no special purpose registers have been changed,*
4. *the allocated addresses of heap variables of the old C0 configuration have not been changed, and*
5. *newly allocated heap variables are (in the* new *assembly machine) allocated directly behind the topmost address of the* old *heap memory.*

$$
\begin{array}{l}
[\![\mathsf{valid\text{-}C0\text{-}conf}\ tt\ pt\ xpt\ c;\ \mathsf{valid\text{-}asm}\ asm;\ \mathsf{allocation\text{-}succeeds}\ tt\ enough\text{-}heap\ c; \\
\ \mathsf{sufficient\text{-}memory}\ max\text{-}address\ tt\ pt\ c;\ \mathsf{sufficient\text{-}memory}\ max\text{-}address\ tt\ pt\ c'; \\
\ \mathsf{consistent}\ tt\ pt\ c\ alloc\ asm;\ \delta\ tt\ pt\ enough\text{-}heap\ c = \lfloor c' \rfloor;\ [\![\lceil c.\mathsf{prog} \rceil]\!] \neq [\,]; \\
\ \neg\ \mathsf{is\text{-}Asm}\ (\mathsf{hd}\ \lceil c.\mathsf{prog} \rceil);\ max\text{-}address \leq 2^{32}]\!] \\
\Longrightarrow \exists\, n\ alloc'\ asm'. \\
\qquad \mathsf{code\text{-}range}\ tt\ (\mathsf{gm\text{-}st}\ c.\mathsf{mem})\ pt, \mathsf{address\text{-}range}\ max\text{-}address \vdash_{\mathsf{asm}} asm \rightarrow^n asm' \wedge \\
\qquad \mathsf{consistent}\ tt\ pt\ c'\ alloc'\ asm' \wedge asm'.\mathsf{sprs} = asm.\mathsf{sprs} \wedge \\
\qquad (\forall\, g.\ \mathsf{valid\text{-}nameless\text{-}gvar}\ tt\ (\mathsf{symbols}\ c.\mathsf{mem})\ g \longrightarrow alloc'\ g = alloc\ g) \wedge \\
\qquad (\mathsf{is\text{-}PAlloc}\ (\mathsf{hd}\ \lceil c.\mathsf{prog} \rceil) \longrightarrow \mathsf{fst}\ (alloc'\ (\mathrm{GVAR\text{-}HM}\ |\mathsf{hm\text{-}st}\ c.\mathsf{mem}|)) = \mathsf{toph}\ asm)
\end{array}
$$

The proof is by case distinction over the first C0 statement in the program rest. For data and code consistency it suffices to examine this statement and its compiled code in isolation. For control consistency we have to show that the *variable* structure of the program rest (which depends on previously executed statements) corresponds to the *fixed* structure of control instructions in the compiled code. We prove this in a separate lemma by induction on the nesting depth of control statements. For details see [Lei08].

5.3 Simulation of C0 by VAMP ISA

In this section we combine the compiler theorem with the VAMP simulation theorem. Again, we only consider system mode executions.

Definition 42 (Valid system mode assembly configuration) An assembly configuration is a valid system mode configuration if it is valid and both the mode and the status register are zero:

$$\textsf{valid-asm-system}\ asm \equiv \textsf{valid-asm}\ asm \wedge asm.\textsf{sprs}[\textsf{MODE}] = 0 \wedge asm.\textsf{sprs}[\textsf{SR}] = 0$$

Definition 43 (Relation between C0 and VAMP ISA configurations) The relation sim-C0-isa, which connects a C0 and a VAMP ISA configuration, holds if there exists a valid system mode VAMP assembly machine *asm* that encodes the C0 configuration *c* via the allocation function *alloc* and is equivalent with the VAMP ISA configuration *isa*, where the program range is computed from *c*.

$$\begin{array}{l}
\textsf{sim-C0-isa}\ tt\ pt\ c\ isa\ alloc \equiv \\
\ \ \exists\, asm.\ \textsf{valid-asm-system}\ asm \wedge \textsf{consistent}\ tt\ pt\ c\ alloc\ asm \wedge \\
\ \ \ \ \textsf{equiv-asm-isa-with-instr}\ asm\ isa\ \textsf{program-base-word}\ (\textsf{codesize-program}\ tt\ (\textsf{gm-st}\ c.\textsf{mem})\ pt)
\end{array}$$

Theorem 7 (Simulation of C0 by VAMP ISA) *Using the abstraction relation* sim-C0-isa *we can combine Theorem 4 with an extended version of Theorem 6 which argues not about a single but n steps of the C0 step semantics starting in configuration c.*

$$\begin{array}{l}
[\![\textsf{valid-C0-conf}\ tt\ pt\ xpt\ c;\ \forall\, i{<}n.\ \neg\ \textsf{is-Asm}\ (\textsf{hd}\ \lceil(\textsf{the}\ (\delta^{i}\ tt\ pt\ enough\text{-}heap\ c)).\textsf{prog}\rceil);\\
\ [\![c.\textsf{prog}]\!] \neq [];\ \textsf{sim-C0-isa}\ tt\ pt\ c\ isa\ alloc;\ \delta^{n}\ tt\ pt\ enough\text{-}heap\ c = \lfloor c'\rfloor;\\
\ \forall\, i{<}n.\ \textsf{allocation-succeeds}\ tt\ enough\text{-}heap\ (\textsf{the}\ (\delta^{i}\ tt\ pt\ enough\text{-}heap\ c)) \wedge\\
\ \ \ \ \textsf{sufficient-memory}\ max\text{-}address\ tt\ pt\ (\textsf{the}\ (\delta^{i}\ tt\ pt\ enough\text{-}heap\ c));\\
\ \textsf{sufficient-memory}\ max\text{-}address\ tt\ pt\ c';\ \textsf{max-address-precondition}\ max\text{-}address;\\
\ \forall\, x \in \textsf{set}\ (\textsf{init-code}\ tt\ (\textsf{gm-st}\ c.\textsf{mem})\ pt).\ \textsf{is-instr}\ x;\ \textsf{precond-seq-isa}\ seq_{\textsf{isa}}\ eifis\ devs]\!]\\
\Longrightarrow \exists\, N\ isa'\ devs'\ alloc'.\\
\qquad \textsf{fst}\ (\Delta_{\textsf{isa}}^{N}\ seq_{\textsf{isa}}\ eifis\ (\!|\textsf{proc} = isa,\ \textsf{devs} = devs|\!)) = (\!|\textsf{proc} = isa',\ \textsf{devs} = devs'|\!) \wedge\\
\qquad \textsf{is-only-changed-mem}\ isa.\textsf{mm}\ isa'.\textsf{mm}\\
\qquad\ (\textsf{program-base} + 4 * \textsf{codesize-program}\ tt\ (\textsf{gm-st}\ c.\textsf{mem})\ pt)\ max\text{-}address \wedge\\
\qquad \textsf{isa-SPRs-equal}\ isa.\textsf{sprs}\ isa'.\textsf{sprs} \wedge c' \in \textsf{valid-C0SS}\ tt\ pt\ \textsf{valid}_{\textsf{asm}}\ xpt \wedge\\
\qquad \textsf{sim-C0-isa}\ tt\ pt\ c'\ isa'\ alloc' \wedge\\
\qquad (\forall\, g.\ \textsf{valid-nameless-gvar}\ tt\ (\textsf{symbols}\ c.\textsf{mem})\ g \longrightarrow alloc'\ g = alloc\ g) \wedge\\
\qquad \textsf{untouched-devs}\ devs\ devs'\ eifis\ seq_{\textsf{isa}}\ N
\end{array}$$

6 CVM

Communicating virtual machines (CVM) is a computational model for the abstract kernel, devices, and a bounded number of user processes [GHLP05,IT08]. All components take turns in execution. The abstract kernel is modeled as C0 machine, the user processes are modeled as assembly machines. CVM abstracts from a low-level microkernel, which implements interrupt handling, kernel entry and exit, and a number of special functions called CVM primitives [ST08a, ST08b]. CVM primitives provide mechanisms for process management, interprocess and device communication. The abstract kernel uses these primitives to implement a scheduler and kernel calls, allowing user processes to interact with each other and with devices. The CVM marks the top-level model described in this article. It is the basis for a microkernel and an operating system built on top of it [DDB08, DDWS08, DDW09].

An important part of CVM is the implementation of memory virtualization, i.e., it ensures that each user process can operate on its own, isolated memory. This is implemented via demand paging based on the VAMP's address translation mechanism and a swap disk. The page table, a data structure both accessed by the processor and by software, maintains whether a page is in the swap or the main memory. Whenever a user process accesses a page that is currently swapped out a page-fault interrupt is triggered by the processor. In response, the page-fault handler is invoked, which copies the requested page back to the main memory. To copy pages between the swap disk and main memory the page-fault handler relies on a hard disk driver. Both the handler and the driver are part of the CVM implementation.

In the remainder of this article we aim to establish the correctness of memory virtualization. In this section we define the CVM abstraction relation with respect to the VAMP ISA and state CVM's correctness in terms of a simulation theorem. One important case in the simulation is the correct virtualization of user steps. The corresponding theorem is stated at the end of this section.

Simulation of user steps most importantly depends on the correctness of the page-fault handler. In Section 7 we first present the top-level page-fault handler correctness statement as it is applied in the user step simulation. This theorem is based on the functional correctness of the paging algorithm, which is presented subsequently. Whereas the CVM implementation is written in C0 with inline assembly and overall CVM correctness is stated at the level of the VAMP ISA, the page-fault handler is basically a sequential C0 program without assembly parts. Thus, we can prove functional page-fault handler correctness in the C0 Hoare logic and then transfer this result down to the VAMP ISA, by utilizing the main theorems presented previously: property transfer from the Hoare logic down to the C0 small-step semantics, compiler correctness, and transfer from VAMP assembly with devices to VAMP ISA with devices.

In the Hoare logic, calls of the page-fault handler to the disk driver are represented as XCalls operating on an extended state, which represents the contents of the swap disk and user memory, i.e., the portion of the main memory reserved for user processes. In Section 8 we prove that the assembly implementation of the driver calls adheres to this abstraction. This is done by a simulation theorem between a C0 machine with XCalls and its implementation in VAMP assembly.

6.1 CVM specification

A CVM configuration cvm is a record with the following components: the device configurations $cvm.\mathsf{devs}$, the interrupt mask $cvm.\mathsf{up.statusreg}$ for the devices, the user process configurations $cvm.\mathsf{up.userprocesses}$, modeled as a map from process IDs to VAMP assembly configurations, the current process identifier $cvm.\mathsf{up.currentp}$, which is $\lfloor pid \rfloor$ if process pid is running and $\perp$ if the kernel is running, and the configuration

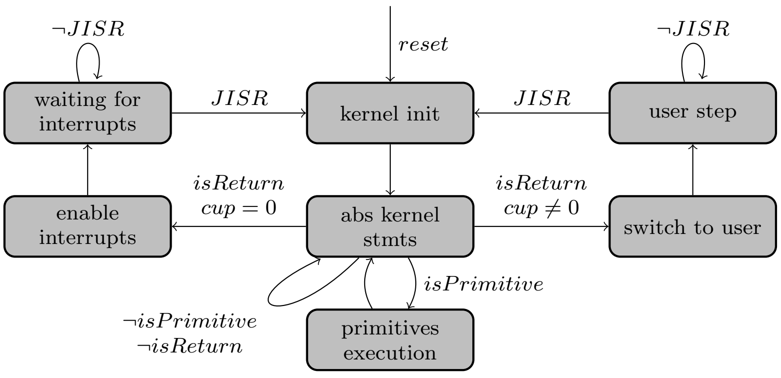


Fig. 4 CVM automaton

$cvm.\mathsf{kernel}$ of the abstract kernel. The abstract kernel is modeled by the C0 small-step semantics. Its configuration includes the type table and the procedure table.

The program of the abstract kernel and the initial device configurations are parameters of the CVM model. In the following we denote them with Π_{k} and $devs$.

Definition 44 (CVM initial configuration) Given the parameters Π_{k} and $devs$, an initial CVM configuration is constructed by the function $\mathsf{init\text{-}cvm\text{-}sys}\ \Pi_{\mathsf{k}}\ devs = cvm$ as follows:

- The current process identifier $cvm.\mathsf{up.currentp}$ is $\perp$ since the computation starts with a kernel step. The status register $cvm.\mathsf{up.statusreg}$ masks all interrupts except for the illegal, misalignment, page-fault, trap, and timer interrupts.
- Device configurations $cvm.\mathsf{devs}$, except for the hard disk, are taken from $devs$. As the CVM abstracts from the page-fault handler and the disk driver, the hard disk used for swapping is no longer visible (i.e., set to the idle device IDLE-DEV).
- The program (i.e., type table, procedure table, and global symbol table) of the abstract kernel $cvm.\mathsf{kernel}$ is set to Π_{k}. The memory of the abstract kernel is initialized and its program rest is set to be a call to the abstract kernel's dispatcher, with parameters indicating a 'system reset'.
- For each user process $cvm.\mathsf{up.userprocesses}\ i$ the program counters and general-purpose registers are set to zero. Their PTL registers are set to -1, which indicates zero size of the virtual memory. Thus, the main memory can be set arbitrarily. The same holds for all special purpose registers other than PTL.[9]

Definition 45 (CVM computation) The transition $\Delta_{\mathsf{cvm}}\ cvm\ dev_{\mathsf{in}}$ takes a CVM configuration cvm and the external input dev_{in} and computes the next state. In case of faults (caused by the abstract kernel), the computation stops with the error configuration $\perp$. Otherwise depending on the value of dev_{in} and $cvm.\mathsf{up.currentp}$, either the kernel, one user process, or one device progresses into the new configuration $\lfloor cvm' \rfloor$ (for kernel and user processes cf. Figure 4). In the last case the CVM computation additionally produces a device output dev_{out}.

Analogously, the transition $\Delta^{N}_{\mathsf{cvm}}\ seq_{\mathsf{asm}}\ cvm$ takes a CVM configuration cvm, the external input sequence seq_{asm} and a step number N and computes the state reached

[9] A user process cannot access special-purpose registers; we use PTL however to encode the processes' memory size, which can be changed by the abstract kernel with CVM primitives.

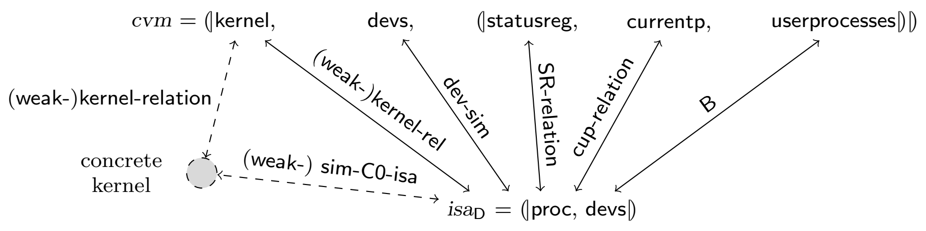


Fig. 5 CVM abstraction relation

after taking N steps. Additionally, an external output list is generated. If any of the steps leads to a fault, the whole computation returns with $\perp$.

6.2 Linking the kernel

The CVM implementation Π_{cvm}, which implements the functions described earlier, is written in C0 with inline assembly. To obtain a running implementation for a given abstract kernel Π_{k}, we link the abstract kernel and the CVM implementation, and compile them for the target machine [IT08], which results in the *concrete kernel*. We merge type and global symbol tables, and replace external function calls to CVM primitives by ordinary calls. We refer to the resulting type, procedure, and (global) symbol table of the concrete kernel by (linked-tt Π_{k}, linked-pt Π_{k}, linked-st Π_{k}).

6.3 Abstraction relation

The abstraction relation maps a CVM configuration to a configuration of VAMP ISA with devices. It is a conjunction of statements for all the components of CVM configurations (cf. Figure 5). We start with the relations SR-relation *cvm*.up.statusreg isa_{D} for the interrupt mask (stored in the status register) and cup-relation (the *cvm*.up.currentp) isa_{D} for the current process identifier. These relations compare the values of the kernel variables *SR* and *cup* obtained from ISA memory with the status register and current process identifier of the CVM configuration. The relation dev-sim *cvm*.devs isa_{D} requires that device states in the CVM and VAMP ISA configurations are equal except for the swap disk, which is an idle device in CVM.

Relation for user processes. The relation B *cvm*.up.userprocesses isa_{D} states whether user process configurations are represented in the configuration of an ISA machine with devices. We first define the function get-p-vm to 'extract' user process configurations from ISA configurations. Second, we define a relation equiv-up which maintains equivalence between the extracted and the given user process configuration.

The function get-p-vm isa_{D} *pid* is defined separately for registers and memory of user process *pid*. If process *pid* is currently running on the ISA machine, its registers can directly be taken from the hardware processor isa_{D}.proc. Otherwise, another process or the abstract kernel is currently running and the registers of process *pid* are held in element *pid* of an array of process control blocks (PCBs). The memory of user process *pid* is stored in the main memory isa_{D}.proc.mm and the swap disk in isa_D.devs. 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. So, (get-p-vm isa_D pid).mm = get-mm isa_D pid, where get-mm is defined as follows:

```
get-mm $isa_D$ $pid$ ≡
  $\lambda ad$. if v (pte $isa_D$ $pid$ (px (4 * $ad$))) = 1
        then get-int-var-from-mem $isa_D$ (pma $isa_D$ $pid$ (4 * $ad$))
        else get-swap-int $isa_D$ (sma $isa_D$ $pid$ (4 * $ad$))
```

The equivalence relation equiv-up compares two user processes asm and asm' with respect to a given (virtual) memory size $vm\text{-}size$. The relation is not an equality: memory contents have to be equal only up to address $vm\text{-}size$, general-purpose register 0 is not compared (it is always tied to 0 for the VAMP), and only certain special-purpose registers must be equal while others are inaccessible to users:

equiv-up asm_1 asm_2 $vm\text{-}size \equiv asm_1.\mathsf{dpc} = asm_2.\mathsf{dpc} \wedge asm_1.\mathsf{pcp} = asm_2.\mathsf{pcp} \wedge$
tl asm_1.gprs = tl asm_2.gprs $\wedge$ ($\forall r \in$set USER-SPRS. $asm_1.\mathsf{sprs}[r] = asm_2.\mathsf{sprs}[r]$) $\wedge$
($\forall ad < vm\text{-}size$ div 4. asm_1.mm ad = asm_2.mm ad)

The relation for a single user process is then defined as

$\mathsf{B_p}$ cvm.up.userprocesses isa_D pid =
equiv-up (get-p-vm isa_D pid) (cvm.up.userprocesses pid) ((ptl isa_D pid + 1) $* 2^{12}$)

where ptl is used to compute the virtual memory size. The conjunction of this relation over all processes defines the user process relation:

B cvm.up.userprocesses isa_D =
$\forall pid.\ 0 < pid \wedge pid <$ PID-MAX $\longrightarrow$ $\mathsf{B_p}$ cvm.up.userprocesses isa_D pid

Relation for the kernel. The abstraction relation for the abstract kernel component cvm.kernel of CVM configurations 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 ck. Concrete and abstract kernels are connected by the relations kernel-relation, which has to hold during kernel executions, or weak-kernel-relation, which has to hold during user steps. Configurations of the concrete kernel ck can be mapped to the architecture by the C0-ISA abstraction relation defined in Section 5.3. Using this relation and some additional invariants to extend the above kernel relations we obtain the combined kernel and weak kernel relations, kernel-rel Π_k and weak-kernel-rel Π_k. Before stating all these relations let us first give an auxiliary definition.

Definition 46 (Content-enclosing relation) For two C0 memory contents ct_1 and ct_2 the predicate shifted-content ct_1 ct_2 lo ho $offset$ len states that the content ct_1 of length len is enclosed within the content ct_2 starting from position $offset$. Local variables are shifted by lo while heap variables are shifted by ho.

Definition 47 (Kernel relation) The relation kernel-relation cvm.kernel Π_k ck is a conjunction of the following facts:

- type-, procedure- and global symbol table of cvm.kernel are equal to those of Π_k,
- the relation kernel-relation-mem cvm.kernel.conf.mem ck.mem which states that the content-enclosing relation holds for the global and heap memories of kernels while the local stack of the abstract kernel is shifted by one (cf. Figure 6),

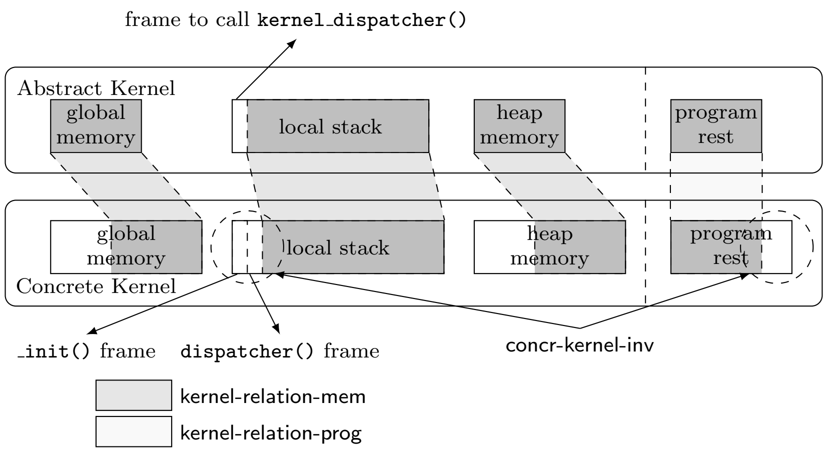


Fig. 6 Relating memory and program rest of abstract and concrete kernel

- the kernel program rest relation kernel-relation-prog *cvm*.kernel.conf Π_k *ck*.mem *ck*.prog which states that both kernels are calling the kernel dispatcher and this call statement in the concrete kernel is obtained by renumbering the statement in the abstract kernel (cf. Figure 6), and
- whenever the abstract kernel terminates its result is equal to the value of the variable *cup* from the concrete kernel.

Definition 48 (Combined kernel relation) The combined kernel relation ties together the abstract kernel *cvm*.kernel and the underlying ISA machine *isa*.

$$\begin{aligned}&\mathsf{kernel\text{-}rel}\ \Pi_k\ cvm.\mathsf{kernel}\ isa\ alloc =\\ &\quad \exists\, pfh_{ss}\ ck.\ \mathsf{kernel\text{-}sim\text{-}C0\text{-}isa}\ \Pi_k\ pfh_{ss}\ ck\ isa\ alloc \wedge \mathsf{concr\text{-}kernel\text{-}inv}\ \Pi_k\ pfh_{ss}\ ck\ \wedge\\ &\quad\ \ \mathsf{kernel\text{-}relation}\ cvm.\mathsf{kernel}\ \Pi_k\ ck\end{aligned}$$

The relation kernel-sim-C0-isa Π_k pfh_{ss} *ck isa alloc* is a conjunction of the C0-ISA abstraction relation, the C0 validity predicate, and properties about the structure of the global and heap memories. The relation concr-kernel-inv Π_k pfh_{ss} *ck* requires that active and free lists and the PCBs in the page-fault handler configuration pfh_{ss} at the small-step layer are well-formed and their values correspond to those in *ck*.

The combined kernel relation has to hold after the execution of every statement of the kernel. However, when the kernel returns control to a user process and until it is re-entered (starting execution again with its main function) this relation is too strong. Instead, we define a weak variant of the kernel relation, which merely requires that the kernel remains 'intact' after user execution, most importantly that its heap and global memory contents remain unchanged. Clearly, the concrete kernel can achieve this by appropriately setting up and restricting address translations for users. In the following definition, *gmct* resp. *hmct* denote global and heap memory contents of the concrete kernel *ck* and *hmst* denotes its heap symbol table.

Definition 49 (Weak kernel relation) The relation weak-kernel-relation requires (i) that type table, procedure table, and global symbol table of *cvm*.kernel equal to those of Π_k and (ii) that the weak kernel memory relation weak-kernel-relation-mem *cvm*.kernel.conf.mem *gmct hmct hmst* holds, which comprises only content-enclosing relations for heap and globals since the stack is reinitialized at the next kernel entry.

As with the kernel relation, we add a C0-ISA abstraction relation (weak, in this case) in order to obtain the combined weak kernel relation.

Definition 50 (Combined weak kernel relation) The combined weak kernel relation maps the abstract kernel cvm.kernel and the underlying ISA machine isa:

```
weak-kernel-rel $\Pi_k$ $cvm$.kernel $isa$ $alloc$ $cp$ =
  $\exists$ $pfh_{ss}$ $gmct$ $hmct$ $hmst$. weak-sim-C0-isa $\Pi_k$ $pfh_{ss}$ $gmct$ $hmct$ $hmst$ $isa$ $alloc$ $\wedge$
    weak-kernel-relation $cvm$.kernel $\Pi_k$ $gmct$ $hmct$ $hmst$ $\wedge$
    fst ($alloc$ (GVAR-HM 0)) = heap-base $\wedge$
    (case $cp$ of $\perp \Rightarrow$ True | $\lfloor pid \rfloor \Rightarrow 0 \leq pfh_{ss}$.abs-pcbs-ss[$pid$].ptl)
```

Besides the weak C0-ISA abstraction relation (explained below) and the weak kernel relation the above definition states that the first variable on the heap resides at the address heap-base and if the current process identifier cp corresponds to a user process, then this process has some allocated memory.

Definition 51 (Weak C0-ISA abstraction relation) The abstraction relation weak-sim-C0-isa Π_k pfh_{ss} $gmcr$ $hmct$ $hmst$ isa $alloc$ states that there exists an intermediate assembly configuration such that (i) it is valid and equivalent to isa, (ii) those parts of the C0 validity predicate hold that speak about global and heap memories, (iii) the compiler consistency relation holds except for control and register consistency.

Putting abstraction relations together. Device, user, and interrupt mask relations hold in every CVM state. Otherwise we distinguish three cases. (i) If cvm.up.currentp $= \perp \wedge$ is-wait-state cvm.kernel the kernel is waiting for an interrupt. In this case the combined weak kernel relation holds and the current process value is equal to 0. (ii) If cvm.up.currentp $= \perp \wedge \neg$ is-wait-state cvm.kernel the kernel is going to make an ordinary step and the full combined kernel relation holds. Additionally, an invariant over processor registers is stated. (iii) If cvm.up.currentp $= \lfloor pid \rfloor$ user process pid is going to make a step. The combined weak kernel relation holds and the value of the current process variable is related to pid. Additionally, the user invariants hold.

```
CVMrelation $cvm$ $\Pi_k$ $isa_D$ $alloc$ $\equiv$ dev-sim $cvm$.devs $isa_D$ $\wedge$ B $cvm$.up.userprocesses $isa_D$ $\wedge$
  SR-relation $cvm$.up.statusreg $isa_D$ $\wedge$
  (case $cvm$.up.currentp of
   $\perp \Rightarrow$ if is-wait-state $cvm$.kernel
        then weak-kernel-rel $\Pi_k$ $cvm$.kernel $isa_D$.proc $alloc$ $\perp$ $\wedge$ cup-relation 0 $isa_D$ $\wedge$
             weak-reg-invariant $isa_D$.proc
        else kernel-rel $\Pi_k$ $cvm$.kernel $isa_D$.proc $alloc$ $\wedge$ reg-invariant $isa_D$.proc
  | $\lfloor pid \rfloor \Rightarrow$ weak-kernel-rel $\Pi_k$ $cvm$.kernel $isa_D$.proc $alloc$ $\lfloor pid \rfloor$ $\wedge$ cup-relation $pid$ $isa_D$ $\wedge$
     user-invariant $isa_D$.proc $pid$)
```

6.4 Correctness theorem

CVM top-level correctness is stated as a simulation theorem between VAMP ISA with devices and the CVM model. We start from an initial ISA configuration (|proc = isa, devs = $devs$|), which is connected to a valid swap disk of sufficient size and has the compiled linked kernel loaded to main memory. The ISA computation is parameterized over an execution sequence seq_{isa}. As a precondition on this sequence, denoted precondition-seq-isa-hd seq_{isa}, we assume that external inputs are well-typed, the processor is scheduled infinitely often, and the hard disk transfers sectors in finite time (this is formalized by an external trigger input to the hard disk signaling progress). Furthermore, we have to assume certain properties of the abstract kernel abs-kernel-props Π_{k}, including preconditions for linking and bounds on the stack size. Based on these assumption we should be able to simulate a CVM computation with an initial state init-cvm-sys Π_{k} *devs* with respect to some execution sequence seq_{asm} (in which devices and the processor always make progress): for all natural numbers n counting non-device steps of the CVM model that reach a state in which the abstract kernel has not caused a fault or heap overflow, we must prove the existence of a step number N' for the ISA model in which, most prominently, the relation CVMrelation holds. Formally:

$$\begin{array}{l}
[\![\text{abs-kernel-props } \Pi_{\mathsf{k}};\ \text{is-init-isa-config } \Pi_{\mathsf{k}}\ isa;\ \text{invariant-hd } devs;\\
\ \text{kernel-size} + \text{max-swap-size} \le \text{hd-size } devs \text{ DID-hd};\ \text{precondition-seq-isa-hd } seq_{\mathsf{isa}}\ eifis\ devs]\!]\\
\Longrightarrow \exists\, seq_{\mathsf{asm}}.\ \text{precondition-seq-cvm } seq_{\mathsf{asm}}\ (\text{init-dev } devs) \wedge\\
\qquad (\forall\, n.\ \exists\, N.\ \text{count-proc-steps } seq_{\mathsf{asm}}\ N = n \wedge\\
\qquad\qquad (\forall\, cvm'\ eifos'.\\
\qquad\qquad\quad \Delta_{\mathsf{cvm}}^{N}\ seq_{\mathsf{asm}}\ (\text{init-cvm-sys } \Pi_{\mathsf{k}}\ devs) = \lfloor (cvm',\ eifos') \rfloor \wedge\\
\qquad\qquad\quad \text{asize-heap } cvm'.\text{kernel.conf.mem.hm.st} \le \text{abstract-heap-max-size} \longrightarrow\\
\qquad\qquad\quad (\exists\, N'\ alloc'.\\
\qquad\qquad\qquad \text{CVMrelation } cvm'\ \Pi_{\mathsf{k}}\ (\text{fst } (\Delta_{\mathsf{isa}}^{N'}\ seq_{\mathsf{isa}}\ eifis\ (\!|\text{proc} = isa,\ \text{devs} = devs|\!)))\ alloc')))
\end{array}$$

The proof of this theorem can be broken down according to the various types of steps that can be made in the CVM model. We focus here on the correct simulation of user processes in the CVM model. The structure of the correctness statement for such steps is similar to the previously stated top-level correctness property. Given that we start computation in a machine configuration $(\!|\text{proc} = isa,\ \text{devs} = devs|\!)$ related to a CVM configuration *cvm* and an abstract kernel Π_{k} via an allocation function *alloc*, we have to show that we can reach a machine configuration $(\!|\text{proc} = isa',\ \text{devs} = devs'|\!)$ in which the user step that the CVM model makes has been simulated.

Theorem 8 (CVM correctness: user step)

$$\begin{array}{l}
[\![\text{abs-kernel-properties } \Pi_{\mathsf{k}};\ \text{invariant-hd } devs;\ \text{precondition-seq-isa-hd } seq_{\mathsf{isa}}\ eifis\ devs;\\
\ \text{kernel-size} + \text{max-swap-size} \le \text{hd-size } devs \text{ DID-hd};\ \text{is-dlx-conft } isa;\ \text{code-invariant-isa } \Pi_{\mathsf{k}}\ isa;\\
\ \text{zfp-condition } isa;\ \text{cvm-stack-and-heap-bound } cvm;\ |(cvm.\text{up.userprocesses } pid).\text{sprs}| = 32;\\
\ \text{CVMrelation } cvm\ \Pi_{\mathsf{k}}\ (\!|\text{proc} = isa,\ \text{devs} = devs|\!)\ alloc;\ cvm.\text{up.currentp} = \lfloor pid \rfloor]\!]\\
\Longrightarrow \exists\, seq_{\mathsf{asm}}.\ \text{precondition-seq-cvm } seq_{\mathsf{asm}}\ (\text{init-dev } devs) \wedge\\
\qquad (\exists\, N.\ \text{count-proc-steps } seq_{\mathsf{asm}}\ N = 1 \wedge\\
\qquad\quad (\forall\, cvm'.\\
\qquad\qquad (\exists\, eifos'.\ \Delta_{\mathsf{cvm}}^{N}\ seq_{\mathsf{asm}}\ cvm = \lfloor (cvm',\ eifos') \rfloor) \longrightarrow\\
\qquad\qquad \text{asize-heap } cvm'.\text{kernel.conf.mem.hm.st} \le \text{abstract-heap-max-size} \longrightarrow\\
\qquad\qquad (\exists\, N'\ isa'\ devs'.\\
\qquad\qquad\quad \text{fst } (\Delta_{\mathsf{isa}}^{N'}\ seq_{\mathsf{isa}}\ eifis\ (\!|\text{proc} = isa,\ \text{devs} = devs|\!)) = (\!|\text{proc} = isa',\ \text{devs} = devs'|\!) \wedge\\
\qquad\qquad\quad (\forall\, DID.\ \text{dconfs-single-same-type } (devs\ DID)\ (devs'\ DID)) \wedge \text{invariant-hd } devs' \wedge\\
\qquad\qquad\quad \text{kernel-size} + \text{max-swap-size} \le \text{hd-size } devs' \text{ DID-hd} \wedge \text{is-dlx-conft } isa' \wedge\\
\qquad\qquad\quad \text{code-invariant-isa } \Pi_{\mathsf{k}}\ isa' \wedge \text{zfp-condition } isa' \wedge\\
\qquad\qquad\quad (\exists\, alloc'.\ \text{CVMrelation } cvm'\ \Pi_{\mathsf{k}}\ (\!|\text{proc} = isa',\ \text{devs} = devs'|\!)\ alloc'))))
\end{array}$$

7 Page-fault handler

Page-faults occurring during a user step are treated by a page-fault handler, a routine which translates addresses and loads missing pages from the hard disk into the physical memory. In this section we report on the verification of the function pfh-touch-addr [ASS08]. This function is called in two situations: when page-fault exceptions occur and when the kernel executes CVM primitives that 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. In this article we concentrate on the first case and omit the remaining details.

7.1 Design and implementation

Our page-fault handler implementation [Con06] maintains several data structures in physical memory to manage physical and swap memories, support virtual memory de- and allocation, and implement a page-replacement strategy. These data structures comprise: (i) the process control blocks (PCBs), described in Section 6.3, (ii) page and big-page tables used for translations of a virtual address into a physical or a swap address, and (iii) active and free lists that manage allocated and free user memory pages. The PCBs are shared between the page-fault handler and the CVM implementation. Page tables are also accessed by the VAMP hardware; therefore, the page table origin address has to match the allocated base address of the C0 variables representing the page table. All other data structures are exclusively accessed by the page-fault handler.

At the software level we distinguish the following page-faults: (i) an *invalid access* page-fault occurs on an access to a page not present in physical memory and (ii) a *zero-protection* page-fault occurs on a write access to a freshly allocated page.

When a memory page is allocated for a user process it must be filled with zeros. This is done lazily by our implementation. During allocation, all freshly allocated pages are mapped read-only to a special *zero-filled page* residing at page address ZFP. Reading from that page returns zero data. At a write attempt a zero-protection page-fault is raised and only then a new page is allocated and cleared. Thus, the original allocation operation leaves the active and free lists unchanged, possibly modifying the PCBs and the page-table space in case an adjustment of origins is needed. When freeing memory, the descriptors of the released pages are moved back to the free list, and the corresponding entries in the page tables are invalidated.

On a page-fault the handler behaves as follows. If there is an unused user page in physical memory (i.e., the free list is not empty), it can be mapped in for the page-faulting process. If not, a page from an active list is evicted. The selected page is then either filled with data loaded from the swap disk or with zeros, depending on the kind of page-fault. The page table entry of an evicted page is invalidated while the valid bit of a loaded page is set. We use the FIFO-eviction strategy, which guarantees that the page swapped in during the previous call to the handler will not be swapped out during the current call. This property is crucial for liveness of user instruction execution since a single instruction can cause up to two page-faults on the physical machine – one during the fetch phase, the other during a load/store operation.

7.2 Top-level correctness

Our notion of the page-fault handler top-level correctness is motivated by the CVM correctness theorem. Besides guaranteeing functional correctness the top-level theorem of the page-fault handler must ensure preservation of the CVM abstraction relation for user processes B and other invariants. The main point of the functional correctness statement is absence of page-faults after a call to the handler.

Theorem 9 (Page-fault handler top-level correctness)

$\llbracket \mathsf{hd}\ \lceil c.\mathsf{prog} \rceil =$
$\textsc{SCall}\ \mathit{res}\ \mathsf{pfh\text{-}touch\text{-}addr}$
$[\textsc{VarAcc}\ \mathit{pid\text{-}vn},\ \textsc{VarAcc}\ \mathit{addr\text{-}vn},\ \textsc{Lit}\ (\mathtt{unsigned}\ \mathsf{MM\text{-}SWAP\text{-}IN}),\ \textsc{Lit}\ (\mathtt{unsigned}\ \mathit{cnt})]\ \mathit{sid};$
$\mathit{intention} = \mathsf{MM\text{-}SWAP\text{-}IN};\ \mathsf{toplm\text{-}conditions}\ \Pi_{\mathsf{k}}\ c\ \mathit{res}\ \mathit{addr\text{-}vn}\ \mathit{va}\ \mathit{pid\text{-}vn}\ \mathit{pid};$
$\mathsf{pfh\text{-}touch\text{-}addr\text{-}PRE}_{\mathsf{ss}}\ (\mathsf{linked\text{-}tt}\ \Pi_{\mathsf{k}})\ \mathit{pfh}_{\mathsf{ss}}\ \mathit{abs\text{-}cnt}\ \mathit{pid}\ \mathit{va}\ \mathsf{MM\text{-}SWAP\text{-}IN}\ \mathit{cnt}\ c.\mathsf{mem.gm.ct}$
$c.\mathsf{mem.gm.st}\ c.\mathsf{mem.hm.ct}\ c.\mathsf{mem.hm.st};$
$\mathsf{abs\text{-}kernel\text{-}props}\ \Pi_{\mathsf{k}};\ \mathsf{stack\text{-}and\text{-}heap\text{-}bound}\ c;\ \mathsf{precondition\text{-}seq\text{-}isa\text{-}hd}\ \mathit{seq}_{\mathsf{isa}}\ \mathit{eifis}\ \mathit{devs};$
$\mathsf{invariant\text{-}hd}\ \mathit{devs};\ \mathsf{kernel\text{-}size} + \mathsf{max\text{-}swap\text{-}size} \leq \mathsf{hd\text{-}size}\ \mathit{devs}\ \mathsf{DID\text{-}hd};\ \mathsf{zfp\text{-}condition}\ \mathit{isa};$
$\mathsf{code\text{-}and\text{-}sprs\text{-}invariant\text{-}isa}\ \Pi_{\mathsf{k}}\ \mathit{isa};\ \mathsf{B}\ \mathit{up}\ (\!|\mathsf{proc} = \mathit{isa},\ \mathsf{devs} = \mathit{devs}|\!);$
$\mathsf{kernel\text{-}sim\text{-}C0\text{-}isa}'\ \Pi_{\mathsf{k}}\ c\ \mathit{isa}\ \mathit{alloc}\rrbracket$
$\Longrightarrow \exists\, N\ \mathit{isa}'\ \mathit{devs}'.$
$\quad \mathsf{fst}\ (\Delta_{\mathsf{isa}}^{N}\ \mathit{seq}_{\mathsf{isa}}\ \mathit{eifis}\ (\!|\mathsf{proc} = \mathit{isa},\ \mathsf{devs} = \mathit{devs}|\!)) = (\!|\mathsf{proc} = \mathit{isa}',\ \mathsf{devs} = \mathit{devs}'|\!) \wedge$
$\quad \mathsf{non\text{-}interference\text{-}hd\text{-}isa}\ \mathit{devs}\ \mathit{devs}'\ \mathit{seq}_{\mathsf{isa}}\ \mathit{eifis}\ N \wedge \mathsf{invariant\text{-}hd}\ \mathit{devs}' \wedge$
$\quad \mathsf{kernel\text{-}size} + \mathsf{max\text{-}swap\text{-}size} \leq \mathsf{hd\text{-}size}\ \mathit{devs}'\ \mathsf{DID\text{-}hd} \wedge \mathsf{zfp\text{-}condition}\ \mathit{isa}' \wedge$
$\quad \mathsf{code\text{-}and\text{-}sprs\text{-}invariant\text{-}isa}\ \Pi_{\mathsf{k}}\ \mathit{isa}' \wedge \mathsf{B}\ \mathit{up}\ (\!|\mathsf{proc} = \mathit{isa}',\ \mathsf{devs} = \mathit{devs}'|\!) \wedge$
$\quad (\exists\, c'\ \mathit{alloc}'.\ \mathsf{kernel\text{-}sim\text{-}C0\text{-}isa}'\ \Pi_{\mathsf{k}}\ c'\ \mathit{isa}'\ \mathit{alloc}' \wedge \mathsf{mem\text{-}structure}\ c\ c'\ \Pi_{\mathsf{k}}\ \mathit{res} \wedge$
$\quad\quad c'.\mathsf{prog} = \mathsf{remove\text{-}first\text{-}stmt}\ c.\mathsf{prog} \wedge$
$\quad\quad \mathsf{lc\text{-}var\text{-}val}\ (\mathsf{linked\text{-}tt}\ \Pi_{\mathsf{k}})\ c'.\mathsf{mem}\ \mathit{res} =$
$\quad\quad \mathtt{unsigned}\ (\mathsf{pfh\text{-}touch\text{-}addr\text{-}result\text{-}ss}\ \mathit{pfh}_{\mathsf{ss}}\ \mathit{abs\text{-}cnt}\ \mathit{pid}\ \mathit{va}\ \mathsf{MM\text{-}SWAP\text{-}IN}) \wedge$
$\quad\quad \mathsf{pfh\text{-}touch\text{-}addr\text{-}POST}_{\mathsf{ss}}\ (\mathsf{linked\text{-}tt}\ \Pi_{\mathsf{k}})\ \mathit{pfh}_{\mathsf{ss}}\ \mathit{abs\text{-}cnt}\ \mathit{pid}\ \mathit{va}\ \mathsf{MM\text{-}SWAP\text{-}IN}\ c'.\mathsf{mem.gm.ct}$
$\quad\quad c'.\mathsf{mem.gm.st}\ c'.\mathsf{mem.hm.ct}\ c'.\mathsf{mem.hm.st})$

In the following we explain the assumptions and conclusions. The first condition of the theorem assumes the head of the kernel's program rest to be an invocation of the page-fault handler. The call takes four parameters. The first two are the PID of the process and the address of the page-fault. The third is the intention of the call. When page-faults are handled, it is set to MM-SWAP-IN. In this case, the fourth parameter is ignored; we do not describe other modes of operation here. The context of the call (the top local memory frame of the kernel) must adhere to certain validity properties, e.g., type constraints for the parameters. Those properties are grouped in the predicate toplm-conditions. The functional preconditions (e.g., restrictions on parameter values) of the handler call are collected in the predicate $\mathsf{pfh\text{-}touch\text{-}addr\text{-}PRE}_{\mathsf{ss}}$.

Moreover, the abstract kernel must satisfy static validity properties described in the CVM predicate abs-kernel-props. The next assumption ensures that the stack and heap size of the kernel machine are bounded, and thus, do not overlap with the memory space of user processes:

$\mathsf{stack\text{-}and\text{-}heap\text{-}bound}\ c \equiv$
$\quad \mathsf{asize\text{-}memlist}\ (\mathsf{map\ st}\ (\mathsf{map\ fst}\ c.\mathsf{mem.lm}))\ \mathsf{frame\text{-}header\text{-}size} \leq \mathsf{MAX\text{-}STACK\text{-}PFH} \wedge$
$\quad \mathsf{asize\text{-}heap}\ c.\mathsf{mem.hm.st} < \mathsf{MAX\text{-}HEAP\text{-}PFH}$

When proving correctness in the VAMP ISA with devices model, we do not have to consider all possible computations given by the execution sequences. Rather, it is valid (by hardware construction) to restrict ourselves only to fair sequences, i.e., sequences in which the processor and the devices are live. These conditions are defined in precondition-seq-isa-hd.

The remaining assumptions are invariants and must also hold after termination of the page-fault handler. The invariant invariant-hd states that the swap disk is valid and of sufficient size. The invariant zfp-condition states that the zero-filled page is placed appropriately in the ISA memory. The invariant code-and-sprs-invariant-isa states that the memory of the ISA machine contains the kernel code, system mode is enabled, and all external interrupts are disabled. The CVM abstraction relation B is another invariant of the page-fault handler. Finally, the invariant $\mathsf{kernel\text{-}sim\text{-}C0\text{-}isa}'$ requires that the VAMP ISA machine implements the concrete kernel.

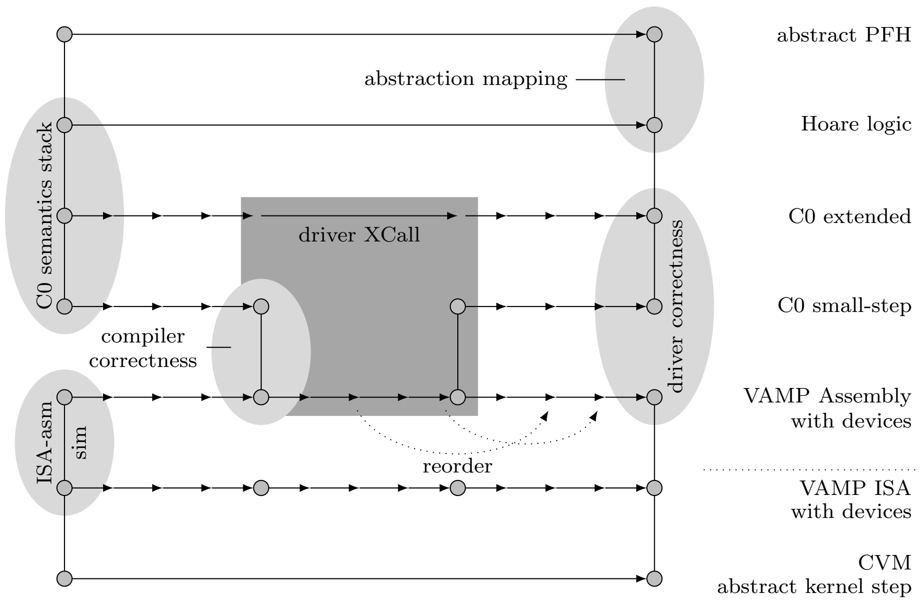


Fig. 7 Putting It All Together – Correctness of the Page-Fault Handler

If all assumptions hold, the theorem claims the existence of a step number N, a corresponding VAMP ISA with devices computation ⦇proc = *isa′*, devs = *devs′*⦈ and a matching new kernel state c', such that the page-fault handler function has returned correctly, the result is written to the respective variable, and the corresponding functional post condition is fulfilled. Moreover, the memory structure of the kernel has been preserved, i.e., only a restricted set of global and heap variables have been altered by the page-fault handler. This is expressed by the predicate mem-structure. For all devices other than the hard disk, we only claim that they have not been influenced by the page-fault handler execution. This notion of non-interference is expressed in the property below, where a transition for an arbitrary device is denoted by the function δ^*_{dev} and the function filter-devs-isa filters out all device steps, which are distinct from the one given:

non-interference-hd-isa *devs devs′* seq_{isa} *eifis* $N \equiv$
  $\forall$ *did*. *did* $\neq$ DID-hd $\longrightarrow$
    devs′ did =
    fst (δ^*_{dev} (map *eifis* (filter-devs-isa seq_{isa} *did* N))
      (replicate |filter-devs-isa seq_{isa} *did* N| mifi-limit-idle) (*devs did*))

Finally, all invariants described above also occur in the conclusion.

7.3 Proof approach

Page-fault handler correctness is ultimately expressed at the level of VAMP ISA with devices (with abstractions provided in the C0 small-step semantics). The page-fault handler implementation consists of C0 and inline assembly portions (e.g., the driver). Thus, it has to be verified with respect to many different semantical layers, forcing us to switch between different verification methods.

The complexity of the proof boils down to obtain the functional postcondition of the page-fault handler at the level of VAMP ISA with devices. Most of the conclusions of the page-fault handler top-level theorem follow from the handler's functional correctness statement $\mathsf{pfh\text{-}touch\text{-}addr\text{-}POST_{ss}}$ which expresses the postcondition of the page-fault handler in terms of the C0 small-step semantics. However, some essential CVM invariants like the relation B are formulated directly in terms of the VAMP ISA. By means of the C0-ISA abstraction relation (cf. Section 5.3) we obtain the values of corresponding memory cells of the underlying ISA machine to infer the conclusions formulated at the ISA level.

The C0 language stack with its extension to XCalls is used to separate verification goals and to apply on each level the adequate technique (cf. Section 3). The overall approach is sketched in Figure 7. To reason on the effects of inline assembly portions in C0, we apply the previously developed concept of XCalls. Recall that each level in the C0 language stack is parametrized over an extended state and an extended semantics. For the verification of the page-fault handler code these parameters are instantiated as follows: First, the extended state is defined, which abstracts from the hard disk and the memory regions not covered by the C0 machine of the kernel. Second, the effect of each inline assembly portion (i.e., for the read and write functions of the hard disk driver) is specified by a separate XCall. Note that both components of the extension are defined almost equally at all levels of the C0 language stack.

Henceforth, we can verify the page-fault handler code solely in extended C0, without resorting to assembly semantics as follows. At the top we introduce a page-fault handler automaton, a HOL abstraction of the handler's data structures. The automaton is used to specify the desired behavior of the handler and to describe necessary validity requirements to it. The automaton's state is mapped to the states of each level of the C0 semantics stack: Simpl, C0 big-step and small-step semantics. By proving in Hoare logic that executing the page-fault handler maintains the abstraction relation between the PFH automaton and the Simpl state, we obtain the desired properties at the level of Simpl. Our next goal is to transfer this correctness result down to the level of ISA exploiting (i) the transfer theorem between Simpl and extended C0 big-step semantics (cf. Section 3.4.1), (ii) the transfer theorem between extended C0 big-step semantics and extended C0 small-step semantics (cf. Section 3.4.2), (iii) the hard disk driver correctness theorem (cf. Section 8), which finally discharges the implementation correctness of the XCalls, and (iv) the VAMP assembly-ISA simulation theorem with device access transfer theorem (cf. Section 4.4). See Section 9.1 for a discussion of how much effort is involved in this process. However, it is quite inconvenient to apply these four theorems directly in the proof of the top-level correctness theorem. Therefore, a number of intermediate theorems were introduced. They state correctness of the page-fault handler at different semantical levels, namely the C0 big-step semantics, the C0 small-step semantics, and the ISA. In the following we show in more details how we establish the page-fault handler's functional correctness at the Hoare logic level.

Automaton and extended state. Functional correctness of the page-fault handler is specified over a page-fault handler automaton and an extended state modeling the physical memory and the swap disk of the machine which runs the handler. The automaton is defined by the record *pfh* and the list of records *pcbs*. The former abstracts data structures from the handler's implementation and has the following fields:

pfh.act	the active list of page descriptors (defined below) associated with user memory pages that store a virtual page
pfh.free	the free list of page descriptors of unused physical pages
pfh.bpfree	the stack of free big-page indices, a list of naturals
pfh.pt	the page-table space used for translating virtual into physical addresses, a two-dimensional list of naturals
pfh.bpt	the big-page table space used for translating virtual into swap addresses, a list of naturals

A page descriptor pd is a record which holds information about one user page and has the following components:

pd.pid	the process identifier associated with the physical page
pd.vpx	the index of the virtual page corresponding to the physical one
pd.ppx	the index of the user page in the physical memory

A single element pcb of the list $pcbs$ is a record collecting those PCB fields that are relevant for page-fault handling:

pcb.pto	page-table origin	pcb.bpto	big-page table origin
pcb.ptl	page-table length	pcb.bptl	big-page table length

The extended state x_{pfh} of the page-fault handler specification is an abstraction of the non-system part of the physical memory of the machine running the handler and the hard disk of this machine. The state is used to specify the effects of read and write operations to / from the hard disk invoked by the handler (cf. Section 8). It has the following components:

x_{pfh}.mem	user part of the physical memory not reachable by C0
x_{pfh}.swap	content of the hard disk excluding the boot region

Both components are two-dimensional lists of natural numbers with dimensions meaning a page content and a page index. We justify the correctness of the mapping of the extended state to the physical memory and the hard disk later (cf. Theorem 11).

The handler must guarantee that after a call to it the page associated with a page-faulting virtual address va of an interrupted user process pid will reside in the physical memory. The page-table entry for a virtual page index vpx computed from the virtual address by means of $\mathsf{px}\ va \equiv va\ \mathsf{div}\ 2^{12}$ is defined as $\mathsf{pte\text{-}pfh}\ pid\ vpx\ pcbs\ pt \equiv pt[\mathsf{pto\text{-}pfh}\ pid\ pcbs + vpx\ \mathsf{div}\ 1024][vpx \wedge_{\mathsf{u}} 1023]$. The function $\mathsf{pto\text{-}pfh}\ pid\ pcbs$ computes the origin of the process pid in the page-table space respecting an offset of the latter in the memory. We extract the valid and protected bits from a page-table entry pte by $\mathsf{v}\ pte$ and $\mathsf{p}\ pte$. While on, these bits denote that the page $\mathsf{px}\ pte$ resides in the physical memory or is read-only, respectively. The handler recognizes a page-fault at a virtual address va of a process pid if $\mathsf{page\text{-}fault}\ pid\ va\ intention\ pcbs\ pt \equiv \mathtt{let}\ pte = \mathsf{pte\text{-}pfh}\ pid\ (\mathsf{px}\ va)\ pcbs\ pt\ \mathtt{in}\ \neg\ \mathsf{v}\ pte \vee intention \neq \mathsf{MM\text{-}READ} \wedge \mathsf{p}\ pte$. The disjuncts denote the invalid access and the zero-protection page-faults, respectively. Since the handler preserves an invariant that all entries pointing to the zero-filled page are protected, the second disjunct does not reference ZFP.

The function $\mathsf{abs\text{-}pfh\text{-}after\text{-}handling}\ \mathit{pfh}\ \mathit{pid}\ \mathit{va}\ \mathit{pcbs}$ specifies the paging algorithm implemented by the handler (cf. Section 7.1). It is invoked on an abstract page-fault handler configuration pfh with $\mathsf{page\text{-}fault}\ \mathit{pid}\ \mathit{va}\ \mathit{intention}\ \mathit{pcbs}\ \mathit{pfh}.\mathsf{pt}$ and returns a non-page-faulting configuration. The update of the extended state, which reflect the page transfers between physical memory and the hard disk, is formalized by the predicate $\mathsf{x\text{-}after\text{-}handling}\ x_{\mathsf{pfh}}\ x_{\mathsf{pfh}}{}'\ \mathit{intention}\ \mathit{pid}\ (\mathsf{px}\ \mathit{va})\ \mathit{pfh}\ \mathit{pcbs}$ which relates initial x_{pfh} and resulting $x_{\mathsf{pfh}}{}'$ configurations of the extended state. The configuration $x_{\mathsf{pfh}}{}'$ is obtained by applying the functions $\mathsf{readPage}$ and $\mathsf{writePage}$ at appropriate page addresses. These functions specify the semantics of the hard disk driver and are described in Section 8.

Validity. We demand various properties to hold over the page-fault handler abstraction. These properties reflect the functional correctness and are necessary for the proof of the relation B. They are established for the first time after the execution of the initialization code and are preserved under calls to the handler. The predicate $\mathsf{valid\text{-}pfh\text{-}pcbs}\ \mathit{pfh}\ \mathit{pcbs}$ claims, among others, the following:

– All virtual addresses are translated into physical ones outside the kernel code range, which is the case if all elements of the page-table space point outside the kernel:

$$\begin{array}{l}\mathsf{valid\text{-}ppx\text{-}ptspace}\ \mathit{pfh} \equiv \\ \quad \forall\, i{<}|\mathit{pfh}.\mathsf{pt}|. \\ \qquad \forall\, j{<}|\mathit{pfh}.\mathsf{pt}[i]|. \\ \qquad\quad \mathsf{v}\ \mathit{pfh}.\mathsf{pt}[i][j] \wedge \mathsf{px}\ \mathit{pfh}.\mathsf{pt}[i][j] \neq \mathsf{ZFP} \longrightarrow \mathsf{KERNEL\text{-}PGS} \leq \mathit{pfh}.\mathsf{pt}[i][j]\ \mathsf{div}\ 2^{12} \wedge \\ \qquad\quad \mathit{pfh}.\mathsf{pt}[i][j]\ \mathsf{div}\ 2^{12} < \mathsf{TOTAL\text{-}PGS}\end{array}$$

– Page tables (of different user processes) do not overlap:

$$\begin{array}{l}\mathsf{pt\text{-}not\text{-}overlap}\ \mathit{pcbs} \equiv \\ \quad \forall\, j{<}|\mathit{pcbs}|.\ \forall\, i{<}j.\ 0 < i \longrightarrow \mathsf{pto\text{-}pfh}\ i\ \mathit{pcbs} + \mathsf{ptl\text{-}pfh}\ i\ \mathit{pcbs} < \mathsf{pto\text{-}pfh}\ j\ \mathit{pcbs}\end{array}$$

– Page-table origins of user processes are monotonic:

$$\mathsf{pto\text{-}mono}\ \mathit{pcbs} \equiv \forall\, j{<}|\mathit{pcbs}|.\ \forall\, i{<}j.\ 0 < i \longrightarrow \mathit{pcbs}[i].\mathsf{pto} < \mathit{pcbs}[j].\mathsf{pto}$$

– The active list describes only valid pages:

$$\mathsf{valid\text{-}bit\text{-}active}\ \mathit{pfh}\ \mathit{pcbs} \equiv \forall\, i{<}|\mathit{pfh}.\mathsf{act}|.\ \mathsf{v}\ (\mathsf{pte\text{-}pfh}\ \mathit{pfh}.\mathsf{act}[i].\mathsf{pid}\ \mathit{pfh}.\mathsf{act}[i].\mathsf{vpx}\ \mathit{pcbs}\ \mathit{pfh}.\mathsf{pt})$$

– All the valid pages are described by the active list:

$$\begin{array}{l}\mathsf{active\text{-}describes\text{-}valid\text{-}pte}\ \mathit{pfh}\ \mathit{pcbs} \equiv \\ \quad \forall\, \mathit{pid}\ \mathit{vpx}. \\ \qquad 1 \leq \mathit{pid} \wedge \mathit{pid} < \mathsf{PID\text{-}MAX} \wedge \mathit{vpx} < \mathsf{to\text{-}nat32}\ (\mathit{pcbs}[\mathit{pid}].\mathsf{ptl} + 1) \wedge \\ \qquad \mathsf{v}\ (\mathsf{pte\text{-}pfh}\ \mathit{pid}\ \mathit{vpx}\ \mathit{pcbs}\ \mathit{pfh}.\mathsf{pt}) \wedge \mathsf{px}\ (\mathsf{pte\text{-}pfh}\ \mathit{pid}\ \mathit{vpx}\ \mathit{pcbs}\ \mathit{pfh}.\mathsf{pt}) \neq \mathsf{ZFP} \longrightarrow \\ \qquad (\exists\, i{<}|\mathit{pfh}.\mathsf{act}|.\ \mathit{pfh}.\mathsf{act}[i].\mathsf{ppx} = \mathsf{px}\ (\mathsf{pte\text{-}pfh}\ \mathit{pid}\ \mathit{vpx}\ \mathit{pcbs}\ \mathit{pfh}.\mathsf{pt}) \wedge \\ \qquad\quad \mathit{pfh}.\mathsf{act}[i].\mathsf{pid} = \mathit{pid} \wedge \mathit{pfh}.\mathsf{act}[i].\mathsf{vpx} = \mathit{vpx})\end{array}$$

– No virtual pages of a given process might be stored by two or more active pages:

$$\mathsf{distinct\text{-}pid\text{-}vpx\text{-}active}\ \mathit{pfh} \equiv \mathsf{distinct}\ (\mathsf{map}\ (\lambda i.\ (i.\mathsf{pid},\ i.\mathsf{vpx}))\ \mathit{pfh}.\mathsf{act})$$

– All physical page indices in active and free lists are distinct:

$$\mathsf{distinct\text{-}ppx\text{-}active\text{-}free}\ \mathit{pfh} \equiv \mathsf{distinct}\ (\mathsf{map}\ \mathsf{ppx}\ (\mathit{pfh}.\mathsf{act}\ @\ \mathit{pfh}.\mathsf{free}))$$

Additionally, a number of size constraints are imposed on extended states:

$$\begin{array}{l}\mathsf{valid\text{-}x}\ x_{\mathsf{pfh}} \equiv |x_{\mathsf{pfh}}.\mathsf{mem}| = \mathsf{USER\text{-}PGS} + 1 \wedge |x_{\mathsf{pfh}}.\mathsf{swap}| = \mathsf{TOTAL\text{-}BIG\text{-}PGS} * 2^{10} \wedge \\ \quad (\forall\, i{<}|x_{\mathsf{pfh}}.\mathsf{mem}|.\ |x_{\mathsf{pfh}}.\mathsf{mem}[i]| = 1024) \wedge (\forall\, i{<}|x_{\mathsf{pfh}}.\mathsf{swap}|.\ |x_{\mathsf{pfh}}.\mathsf{swap}[i]| = 1024)\end{array}$$

Abstraction towards Simpl. To state that the handler's implementation respects the algorithm defined over the automaton we introduce an abstraction relation. It ties together the automaton's state and the state of the handler's implementation at each level of the C0 semantics stack. At the Hoare logic level the C0 implementation is translated into Simpl. The state is given as a collection of flattened variables – all structure fields are unrolled – and a separate heap function for every field (split heap approach). The abstraction relation pfh-pcbs-map holds between the variables constituting the Simpl state and the page-fault handler automaton given by *pfh* and *pcbs*. The relation maps each variable from the implementation to its equivalent notion in the automaton. The mapping is non-trivial to a large extent because pointer data structures, like doubly-linked lists, are used. So far we have defined: (i) the page-fault handler automaton and its validity requirements, (ii) the page-fault handling algorithm which operates over the automaton and the extended state, and (iii) the abstraction relation between the automaton and the implementation in Simpl. To guarantee the handler's implementation correctness we have proven in the Hoare logic the following theorem.

Theorem 10 (Page-fault handler functional correctness) *Let pfh and pcbs define the page-fault handler automaton, let* x_{pfh} *be an extended state, let pid be a process identifier, and let* va *be a virtual address. Assume that*

– pfh-pcbs-map *holds between the Simpl pre-state and pfh with pcbs,*
– valid-pfh-pcbs *pfh pcbs holds, and*
– page-fault *pid* va *intention pcbs pfh*.pt *takes place.*

If $pfh' =$ abs-pfh-after-handling *pfh pid* va *pcbs and* $x_{\mathsf{pfh}}{}'$ *is appropriately constructed using* readPage *and* writePage *then*

– pfh-pcbs-map *holds between the Simpl post-state and* pfh' *with pcbs,*
– valid-pfh-pcbs pfh' *pcbs is preserved,*
– $\neg$ page-fault *pid* va *intention pcbs* pfh'.pt*, and*
– x-after-handling x_{pfh} $x_{\mathsf{pfh}}{}'$ *intention pid* (px va) pfh' *pcbs.*

8 Disk driver

Device drivers are an integral part of system software. Not only high-level functionality such as file I/O or networking depend on devices, even basic operating system features, such as demand paging (Sect. 7), rely on correctly implemented device drivers. Hence, any verification approach of computer system stacks should deal with driver correctness. When proving functional correctness of a driver it is insufficient to reason only about software, devices themselves and their interaction with the processor have to be taken into account. In this section we present the correctness theorem of a simple hard disk driver for an ATAPI hard disk, which is reading and writing single pages. The driver is called by the page-fault handler to swap data from and to the hard disk. It is implemented in C0 with inline assembly code; the specification of each driver functions is given in form of an atomic XCall. The correctness of these specifications is expressed in terms of a simulation theorem.[10] With the theorem we present, we

[10] Currently only the write case is formally proven; proving the read case is an ongoing effort.

discharge the implementation correctness proof obligation of the XCalls used in the context of the page-fault handler verification. Moreover, page-fault handler correctness itself gets transferred from C0 with XCalls down to VAMP assembly with devices.

In [ASS08] we addressed the embedding of the hard disk driver into page-fault handler verification. Furthermore, in [AH08] a formal model of an ATAPI hard disk is described and a general theory for driver verification in an interleaved setting is introduced and applied to the concrete hard disk driver.

8.1 Driver specification and abstraction relation

The driver is specified by means of XCalls, which operate on the extended state defined in Section 7.3, in particular its memory component $x_{\mathsf{pfh}}.\mathsf{mem}$ and swap component $x_{\mathsf{pfh}}.\mathsf{swap}$. The two driver XCalls are $\mathsf{writePage}$, which copies a page from the memory to the swap component, and $\mathsf{readPage}$, which does the opposite.

In the following we refer to the pair consisting of the ordinary C0 state and the extended driver component as *extended C0 machine*. Basically, the simulation theorem relates the execution of the extended C0 machine with the execution of the concurrent VAMP assembly with devices model. Furthermore, we maintain on the implementation side an intermediate C0 configuration which has to be consistent with the assembly state. We use the intermediate C0 machine because the extended C0 machine does not cover the whole C0 implementation and hence cannot be related to the underlying assembly state (e.g., the C0 functions implementing the XCalls are not contained in the procedure table).

Before stating the theorem, we define the corresponding abstraction relation. In short, it relates code and configurations of the extended C0 machine with the intermediate C0 machine and VAMP assembly with devices. The code portions of the intermediate C0 machine are basically obtained by implementing the extended procedures by ordinary procedures and replacing every XCall by an ordinary procedure call. This is the core of the following abstraction relation.

Definition 52 (XCall code abstraction relation) The abstraction relation for code $\mathsf{codeSIM}\ pt_{\mathsf{i}}\ p_{\mathsf{i}}\ pt_{\mathsf{x}}\ p_{\mathsf{x}}\ xpt\ specMap$ takes as parameters the procedure tables and program rests of the intermediate and the extended C0 machine, the extended procedure table, and a map relating XCalls to their implementation functions in the intermediate C0 machine. It is the conjunction of three properties:

Program rests. By replacing each occurrence of a driver XCall by an invocation of the corresponding implementation function in the program rest p_{x} of the extended machine we must obtain the program rest p_{i} of the implementation machine.

Procedure tables. All functions defined in the procedure table pt_{x} of the extended machine must also be defined in the procedure table pt_{i} of the implementation machine. Additionally, the functions have to be equal except for their bodies for which the program rest relation from above has to hold.

Relating XCalls and C0 implementation. This property ensures that the intermediate C0 machine implements the XCalls by corresponding C0 functions. The map of XCalls to their implementation is given by the parameter *specMap*, which contains: the names of the XCalls to simulate, the signatures of the C0 functions implementing them, and the semantics of the XCalls.

We define the abstraction relation for the driver in two steps, starting with a slightly simplified version $\mathsf{driverSIM}'$.

Definition 53 (XCall abstraction relation) The abstraction relation for the disk driver $\mathsf{driverSIM}'\ tt\ pt_{\mathsf{i}}\ c_{\mathsf{i}}\ alloc\ asm_{\mathsf{D}}\ pt_{\mathsf{x}}\ c_{\mathsf{x}}\ xpt\ specMap$ takes as parameters the description of the intermediate C0 machine (consisting of type table, procedure table, and current state), the allocation function from the compiler, the current state of the VAMP assembly with devices model, the description of the extended C0 machine (consisting of procedure table, current state, and the extended procedure table) and a map relating the XCall specifications to their implementation in the intermediate C0 machine. It is defined as the conjunction of the following properties:

Extended C0 / VAMP assembly with devices. We map the extended swap component to the sector memory of the hard disk: $\mathsf{memConsis}\ asm_{\mathsf{D}}.\mathsf{proc}.\mathsf{mm}\ (\mathsf{snd}\ c_{\mathsf{x}}).\mathsf{mem}$. The parts of the memory of the assembly machine, which lie outside of the range of the C0 machine are mapped to the abstract memory component of the extended machine; this is captured by $\mathsf{swapConsis}\ (asm_{\mathsf{D}}.\mathsf{devs}\ \mathsf{DID\text{-}hd})\ (\mathsf{snd}\ c_{\mathsf{x}}).\mathsf{swap}$.

Extended C0 / Intermediate C0. The memory of the intermediate C0 machine and the extended machine are equal: $c_{\mathsf{i}}.\mathsf{mem} = (\mathsf{fst}\ c_{\mathsf{x}}).\mathsf{mem}$. Additionally, the code and the procedure tables of the extended C0 machine and of the C0 implementation machine are related: $\mathsf{codeSIM}\ pt_{\mathsf{i}}\ c_{\mathsf{i}}.\mathsf{prog}\ pt_{\mathsf{x}}\ (\mathsf{fst}\ c_{\mathsf{x}}).\mathsf{prog}\ xpt\ specMap$.

Intermediate C0 / VAMP assembly with devices. Compiler consistency holds between intermediate C0 and assembly machine: $\mathsf{consistent}\ tt\ pt_{\mathsf{i}}\ c_{\mathsf{i}}\ alloc\ asm_{\mathsf{D}}.\mathsf{proc}$.

8.2 Driver correctness theorem

With the abstraction relation we can sketch the outline of the simulation theorem. It relates the execution of n steps of the specification machine to N steps in the VAMP assembly with devices model. Remember that the term $\lambda m\ t.\ \mathsf{True}$ is our instantiation for the predicate *enough-heap* (cf. Section 5.2).

$$\begin{aligned}
&[\![\mathsf{driverSIM}'\ tt\ pt_{\mathsf{i}}\ c_{\mathsf{i}}\ alloc\ asm_{\mathsf{D}}\ pt_{\mathsf{x}}\ (c_{\mathsf{x}}, x_{\mathsf{pfh}})\ xpt\ \mathsf{driver\text{-}impl\text{-}to\text{-}spec};\\
&\ \delta_{\mathsf{x}}^{n}\ tt\ pt_{\mathsf{x}}\ (\lambda m\ t.\ \mathsf{True})\ c_{\mathsf{x}}\ x_{\mathsf{pfh}}\ xpt = \lfloor (c_{\mathsf{x}}{}', x_{\mathsf{pfh}}{}') \rfloor]\!]\\
&\Longrightarrow \exists\, c_{\mathsf{i}}{}'\ alloc'\ N.\\
&\qquad \mathsf{driverSIM}'\ tt\ pt_{\mathsf{i}}\ c_{\mathsf{i}}{}'\ alloc'\ (\mathsf{fst}\ (\Delta_{\mathsf{asm}}^{N}\ seq\ asm_{\mathsf{D}}))\ pt_{\mathsf{x}}\ (c_{\mathsf{x}}{}', x_{\mathsf{pfh}}{}')\ xpt\ \mathsf{driver\text{-}impl\text{-}to\text{-}spec}
\end{aligned}$$

This is the granularity at which theorems usually are presented in conference articles (e.g., in [ASS08]): the skeleton of the simulation is sketched, where the bulk of complex but presumably dreary conditions are silently omitted. Those conditions, though, are often highly intriguing, hard to predict, and unveil only during the formal verification process. Staying with the skeleton picture, they are the meat of our work.

In the following we extend the skeleton to a full-blown version of the driver correctness, as it has been proven in Isabelle and applied in the formal verification of the page-fault handler. Most of the assumptions and conclusions appearing in the driver correctness are already defined and originate either from the compiler, the page-fault handler, or CVM correctness.

Compositionality conditions. The driver correctness theorem serves to translate correctness results from the level of C0 with XCalls down to the VAMP assembly with devices model, in which the assembly state and devices are explicitly visible. Applied to the stack presented in this article, the theorem is used to propagate the correctness of the page-fault handler to the level of VAMP assembly with devices. This result is then embedded into the verification of the CVM.

However, the outlined theorem does not allow such an embedding of a smaller computation (page-fault handler) into a larger computation (CVM), because the assumption on the program rest is too restrictive. It requires the code of the extended C0 machine and that of the implementation machine to be equal (up to XCall substitution). This is a major restriction: the page-fault handler is verified relative to an extended machine only containing the functions needed for its invocation not aware of the rest of the CVM code.

The problem is close to the one described and solved by Theorem 3. Unfortunately, this theorem does not serve us well: page-fault handler correctness is embedded into CVM correctness at the level of the VAMP rather than at the pure C0 level. Thus, we generalize the driver correctness theorem to obtain a modular verification chain. (i) Similar to the property transfer in the C0 language stack we propagate correctness at the level of function calls: $\mathsf{is\text{-}SCall}\ c_{\mathsf{x}}.\mathsf{prog}$. (ii) Because the extended machine is embedded into the implementation machine, the procedure table of the former must be included in the one of the latter. This is already guaranteed by the codeSIM relation. (iii) The program rest of the extended machine is only a prefix of the implementation machine (after XCall substitution). Thus, we obtain the new abstraction relation driverSIM by refining the code relation of the abstraction relation driverSIM′: $\exists\, p_{\mathsf{pre}}.\ \mathsf{prefix}\ c_{\mathsf{i}}.\mathsf{prog}\ p_{\mathsf{pre}}\ rest \wedge \mathsf{codeSIM}\ pt_{\mathsf{i}}\ p_{\mathsf{pre}}\ pt_{\mathsf{x}}\ p_{\mathsf{x}}\ xpt\ specMap.$

Validity predicates. We have to assume validity of the C0 configuration, the extended C0 configuration, the assembly state, the hard disk, and the execution sequences of the VAMP assembly with devices model.

To apply C0 compiler correctness, validity of the initial C0 machine has to be assumed (see Section 5): $\mathsf{valid\text{-}C0\text{-}conf}\ tt\ pt_{\mathsf{i}}\ []\ c_{\mathsf{i}}$.

Inline assembly code in the extended machine may break the abstraction provided by XCalls. For example, a code portion writing the abstracted memory region would have no semantical effects on the abstract memory component and hence lead to an unsound state. Therefore, we require the program rest of the extended machine and of the bodies of all functions $fn \in \mathsf{SCalls}\ xpt\ (\mathsf{scalls}\ xc.\mathsf{prog})$ possibly called during its execution to be free of inline assembly code:

$\mathsf{precondition\text{-}C0X\text{-}driver}\ pt_{\mathsf{x}}\ c_{\mathsf{x}} \equiv$
$\quad \forall\, fn \in \mathsf{SCalls}\ pt_{\mathsf{x}}\ (\mathsf{scalls}\ c_{\mathsf{x}}.\mathsf{prog}).$
$\qquad \mathtt{case}\ \mathsf{map\text{-}of}\ pt_{\mathsf{x}}\ fn\ \mathtt{of}\ \bot \Rightarrow \mathsf{True} \mid \lfloor pte \rfloor \Rightarrow \mathsf{valid\text{-}prop}\ pte.\mathsf{proc\text{-}body}\ \mathsf{noASM}$

The device component is valid if a hard disk is connected that is in a valid state. A hard disk is in a valid state if it is idle (i.e., no operation is pending), if interrupts are disabled, if its sector memory is well defined, i.e., the values of the data stored are in range, and if its buffer is empty.

Execution sequences define for each step, whether the processor or one of the devices is allowed to take action. For unfair sequences, i.e., if either the hard disk or the processor are scheduled only finitely many times, we cannot prove termination and thus also no driver correctness. However, from hardware correctness one can deduce that it suffices only to consider fair sequences. Furthermore, we need to assume that the sequences are well-typed, i.e., that the type of the external input always matches the type of the device taking a step:

$\mathsf{precondition\text{-}seq\text{-}asm}\ seq_{\mathsf{asm}} \equiv$
$\quad \mathsf{eifis\text{-}welltyped\text{-}asm}\ seq_{\mathsf{asm}} \wedge \mathsf{proc\text{-}live\text{-}input\text{-}seq\text{-}asm}\ seq_{\mathsf{asm}} \wedge \mathsf{hd\text{-}live\text{-}input\text{-}seq\text{-}asm}\ seq_{\mathsf{asm}}$

The next predicate ensures that certain components of the system did not change during execution. This holds for the length of the disk memory, the length of the abstract swap component, and for the first two memory cells of the assembly machine (containing a jump instruction to the kernel code). The last conjunct is part of the predicate kernel-sim-C0-isa (cf. page 35). It ensures that once the page tables are created, their allocation in the processor memory will not change:

$\mathsf{invariant\text{-}mem}\ asm_{\mathsf{D}}\ asm_{\mathsf{D}}'\ c_{\mathsf{i}}\ c_{\mathsf{i}}'\ alloc\ alloc'\ x_{\mathsf{pfh}}\ x_{\mathsf{pfh}}' \equiv$
$\quad \mathsf{hd\text{-}size}\ asm_{\mathsf{D}}.\mathsf{devs}\ \mathsf{DID\text{-}hd} = \mathsf{hd\text{-}size}\ asm_{\mathsf{D}}'.\mathsf{devs}\ \mathsf{DID\text{-}hd} \wedge$
$\quad |x_{\mathsf{pfh}}'.\mathsf{swap}| = |x_{\mathsf{pfh}}.\mathsf{swap}| \wedge$
$\quad (\forall\, i.\ 0 \leq i \wedge i < 2 \longrightarrow asm_{\mathsf{D}}'.\mathsf{proc.mm}\ i = asm_{\mathsf{D}}.\mathsf{proc.mm}\ i) \wedge$
$\quad (c_{\mathsf{i}}'.\mathsf{mem.hm.st} \neq [] \longrightarrow$
$\quad\ \ \mathsf{fst}\ (alloc'\ (\textsc{gvar-hm}\ 0)) =$
$\quad\ \ (\mathtt{if}\ c_{\mathsf{i}}.\mathsf{mem.hm.st} = []\ \mathtt{then}\ \mathsf{toph}\ asm_{\mathsf{D}}.\mathsf{proc}\ \mathtt{else}\ \mathsf{fst}\ (alloc\ (\textsc{gvar-hm}\ 0))))$

Translation from VAMP assembly to VAMP ISA. The assembly model is only an abstraction (and simplification) of the underlying ISA. For translating computations from VAMP assembly with devices to VAMP ISA with devices via Theorem 5, we need to discharge a series of assumptions on the initial state and assumptions on each step of the execution. The former is given by the already introduced predicate valid-asm-system (cf. Section 5.3). Among others, the dynamic conditions forbid instructions for unmasking external interrupts and for switching the processor mode. They are grouped together by the already defined predicate dynamic-properties-dev (cf. Section 4.4).

Memory restrictions. The compiler correctness theorem is only applicable if in each step sufficient heap and stack memory is available in the assembly machine (see Section 5). Often such assumptions are silently ignored because they are not visible in the semantics of the given high-level language. As in the C0 small-step semantics, those models assume some infinite memory. Memory restrictions do not emerge until results are propagated down to lower levels, as in the case of the driver correctness.

If no garbage collection is running the heap consumption of a C0 program will steadily increase. The reason is simple: C0 offers only a construct to allocate new memory on the heap, but not to set memory free again. Thus, it suffices to claim the sufficient-heap predicate only for the final configuration to ensure that sufficient heap is available during all predecessor computations:

$\mathsf{heap\text{-}base} + \mathsf{asize\text{-}heap}\ c_{\mathsf{x}}'.\mathsf{mem.hm.st} < max\text{-}address$

Similarly, we would like to estimate the maximum stack consumption needed by the considered computation. An upper bound of the stack consumption of functions with no recursive calls can be computed by a static analysis of the program. In short, this approach determines the deepest path (in terms of stack consumption) in the invocation tree of the given code. This path is computed by the recursively defined set sufficient-stack-size. A triple (pt_{i}, fn, sz) is an element of this set if sz is an upper bound of the stack consumption required of the function fn in the procedure table pt_{i}. As mentioned before, we propagate correctness at the level of function calls (as for example the call to the page-fault handler). For such a function call we get the size estimation sz by:

$(pt_{\mathsf{i}},\ \mathsf{the}_{\textsc{called-function}}\ (\mathsf{hd}\ \lceil c_{\mathsf{x}}.\mathsf{prog}\rceil),\ sz) \in \mathsf{sufficient\text{-}stack\text{-}size}$

We require that the sum of this upper bound and the current stack consumption to be smaller than the heap-base:

$sz + \text{abase-local-frame } tt\ pt_{\mathrm{i}}\ (\text{symbols } c_{\mathrm{i}}.\text{mem})\ (\text{recursion-depth } c_{\mathrm{i}}.\text{mem}) \leq \text{heap-base}$

The next condition ensures that the driver code is running within the kernel memory, i.e., that neither user pages are manipulated by the C0 code nor that the C0 code accesses any devices through memory mapped I/O:

$max\text{-}address \leq (\text{KERNEL-PGS} - 1) * \text{PAGE-SIZE}$

All memory restrictions are grouped in the predicate sufficient-memory-driver. In terms of our application stack, sufficient-memory-driver has been discharged for the C0 machine of the concrete kernel.

Interleaving devices. So far we only described the semantical effects on the hard disk, silently ignoring all other devices. Because the code only accesses the hard disk, the computation of other devices is not influenced by the simulated computation of the processor. Still, due to input from the external environment, the configuration of the devices may have changed. Thus, the final state of any device can be computed by ignoring all steps of the processor and of the hard disk. This notion of non-interference is expressed in the property below (cf. Section 7.2).

$$
\begin{array}{l}
\text{non-interference-hd-asm } asm_{\mathrm{D}}\ seq\ i \equiv \\
\quad \forall\, did.\ did \neq \text{DID-hd} \longrightarrow \\
\qquad\quad (\text{fst } (\Delta^{i}_{\mathrm{asm}}\ seq\ asm_{\mathrm{D}})).\text{devs } did = \\
\qquad\quad \text{fst } (\delta^{*}_{\mathrm{dev}}\ (\text{filter-devs-asm } seq\ did\ i) \\
\qquad\qquad\quad (\text{replicate } |\text{filter-devs-asm } seq\ did\ i|\ \text{mifi-limit-idle})\ (asm_{\mathrm{D}}.\text{devs } did))
\end{array}
$$

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

Theorem 11 (Driver correctness)

$$
\begin{array}{l}
[\![\text{valid-C0-conf } tt\ pt_{\mathrm{i}}\ []\ c_{\mathrm{i}};\ \text{valid-asm-system } asm_{\mathrm{D}}.\text{proc};\ \text{invariant-hd } asm_{\mathrm{D}}.\text{devs}; \\
\ \text{precondition-seq-asm } seq_{\mathrm{asm}};\ \text{precondition-C0X-driver } pt_{\mathrm{x}}\ c_{\mathrm{x}}; \\
\ \text{sufficient-memory-driver } tt\ pt_{\mathrm{i}}\ c_{\mathrm{i}}\ c_{\mathrm{x}}\ c_{\mathrm{x}}'\ max\text{-}address\ sz;\ \text{is-SCall } c_{\mathrm{x}}.\text{prog}; \\
\ \text{driverSIM } tt\ pt_{\mathrm{i}}\ c_{\mathrm{i}}\ rest\ alloc\ asm_{\mathrm{D}}\ pt_{\mathrm{x}}\ (c_{\mathrm{x}},\ x_{\mathrm{pfh}})\ xpt\ \text{driver-impl-to-spec}; \\
\ \delta^{k}_{\mathrm{x}}\ tt\ pt_{\mathrm{x}}\ (\lambda x\ y.\ \text{True})\ c_{\mathrm{x}}\ x_{\mathrm{pfh}}\ xpt = \lfloor(c_{\mathrm{x}}',\ x_{\mathrm{pfh}}')\rfloor]\!] \\
\Longrightarrow \exists\, c_{\mathrm{i}}'\ alloc'\ i.\ \text{valid-C0-conf } tt\ pt_{\mathrm{i}}\ []\ c_{\mathrm{i}}' \wedge \\
\qquad \text{valid-asm-system } (\text{fst } (\Delta^{i}_{\mathrm{asm}}\ seq_{\mathrm{asm}}\ asm_{\mathrm{D}})).\text{proc} \wedge \\
\qquad \text{invariant-hd } (\text{fst } (\Delta^{i}_{\mathrm{asm}}\ seq_{\mathrm{asm}}\ asm_{\mathrm{D}})).\text{devs} \wedge \\
\qquad \text{invariant-mem } asm_{\mathrm{D}}\ (\text{fst } (\Delta^{i}_{\mathrm{asm}}\ seq_{\mathrm{asm}}\ asm_{\mathrm{D}}))\ c_{\mathrm{i}}\ c_{\mathrm{i}}'\ alloc\ alloc'\ x_{\mathrm{pfh}}\ x_{\mathrm{pfh}}' \wedge \\
\qquad \text{dynamic-properties-dev } asm_{\mathrm{D}}\ (\text{program-base div } 4) \\
\qquad\ (\text{codesize-program } tt\ (\text{gm-st } c_{\mathrm{i}}.\text{mem})\ pt_{\mathrm{i}})\ seq_{\mathrm{asm}}\ i \wedge \\
\qquad \text{non-interference-hd-asm } asm_{\mathrm{D}}\ seq_{\mathrm{asm}}\ i \wedge \\
\qquad \text{driverSIM } tt\ pt_{\mathrm{i}}\ c_{\mathrm{i}}'\ rest\ alloc'\ (\text{fst } (\Delta^{i}_{\mathrm{asm}}\ seq_{\mathrm{asm}}\ asm_{\mathrm{D}}))\ pt_{\mathrm{x}}\ (c_{\mathrm{x}}',\ x_{\mathrm{pfh}}')\ xpt \\
\qquad\ \text{driver-impl-to-spec}
\end{array}
$$

8.3 Proof methodology and overview

Obviously, when proving correctness of a concrete driver, an interleaved semantics of all devices is extremely cumbersome. Integration of results into traditional Hoare logic proofs also becomes hardly manageable. Preferably, we would like to maintain a sequential programming model or at least, only bother with interleaved steps of those devices controlled by the driver we attempt to verify. This can be achieved by exploiting non-interference properties of the processor and devices.

In Section 8.2 we stated non-interference of processor computations on devices not accessed by the code. Similarly, we can also show the opposite direction: The computation of a processor that only accesses the hard disk is not influenced by other devices in case external interrupts are disabled. Thus, when verifying our driver it suffices to consider only those execution sequences which are restricted to processor and hard disk steps.

In certain *stable states* the hard disk is guaranteed not to be subject of manipulation due to external input. During this time, reasoning can be simplified even more and a completely sequential model in which only the processor takes steps can be assumed.

The sketched non-interference and reordering lemmas are detailed and applied to the verification of the hard disk driver in [AH08]. The proof of the theorem consists of two major parts: the correctness of the inlined assembly code and its correct embedding into a C0 function call. The assembly code copies a page sector by sector to the internal memory of the hard disk. Each sector is first written word by word to an internal buffer of the disk. Once the buffer is full it takes the disk some time until it copies the data to the sector memory. Meanwhile, the driver polls on a dedicated status port. The end of copying is indicated by a (non-deterministic) input from the external environment.

The correctness of the driver is expressed in terms of the interleaved assembly model with devices. Thus, all valid execution sequences have to be considered. However, by knowing that the disk is almost always in a stable mode, and by using the non-interference observation described above, large parts of the proof could be carried out in a sequential model, in which only the processor takes steps. The only place were some concurrent reasoning is necessary is during the polling loop.

The assembly code is embedded into a C0 function call. However, this call must finally also be expressed in terms of the underlying assembly model with devices. This is accomplished by several applications of the C0 compiler correctness theorem. Compiler correctness is still applicable in the context of interleaved executed devices. This follows from the non-interference observation stated above, and since (i) the compiler ensures that the compiled code does not access any device ports, and (ii) the disk remains in stable configurations during execution of C0 statements.

After the C0 function is invoked, parameters are passed to a freshly created function frame. Compiler correctness ensures correctness of parameter passing and that the program counter of the compiled assembly machine is pointing to the beginning of the assembly driver code. Now, we apply the previously sketched correctness of the inlined code and obtain a new assembly configuration. We need to show that meanwhile, the C0 abstraction is not broken, e.g., by manipulations of stack or heap pointers. Finally, again by using compiler correctness, we show that the function returns correctly.

9 Experiences and lessons learned

9.1 Statistics and evaluation

Table 3 summarizes some statistics of the theories, arranged along the sections of this article. The numbers are based on Isabelle's keywords, instead of raw 'lines of code'. As with all statistics, the numbers have to be interpreted with care, especially regarding the proof steps. The theories were developed by various people from different sites with different backgrounds and Isabelle offers its users quite some opportunity to develop an individual proof style. Some prefer rather small steps others try to push automation as far as possible. Some prefer the 'apply' style, which often leads a large number of intermediate lemmas, others structure their proofs with Isar.

Table 3 Theory statistics

Module	Section	Definitions	Lemmas	Proof steps
C0 (syntax)	3	178	158	903
Hoare logic	3.1	95	866	15 748
C0 big-step	3.2	100	307	7 691
C0 small-step	3.3	357	1 212	19 482
Hoare to big-step	3.4.1	118	591	16 713
Big-step to small-step	3.4.2	88	318	16 492
VAMP	4	877	1 366	21 419
C0 compiler	5	282	1 402	41 234
CVM	6	1 223	3 231	53 536
Page-fault handler	7	453	1 577	31 783
Disk driver	8	162	411	11 908
Miscellaneous		166	828	7 626
Σ		4 099	12 267	244 535

The statistics illustrate that pervasive verification comes at a cost. The development of the meta theory and the utilization of the theorems to actually transfer properties are expensive. However, to do justice to the numbers one has to keep in mind that the meta theorems only have to be proven once and there is a lot of potential for automation to support the application of the theorems for actual property transfer. We have only developed this kind of automation to limited degree, as we only transferred a couple of properties within the time limits of the Verisoft project. After all the overhead for pervasiveness can be considered constant the more properties are actually transferred.

The main benefit of pervasive verification and a seamlessly verified model stack with a small trusted computing base are that an extreme level of confidence and assurance is gained in specifications and results. There is no fear of redoing parts of your work as soundness bugs become apparent, which is the ‘Sword of Damocles’ hanging over all less rigorous approaches. To make pervasive verification more efficient we recommend to focus future effort on reusable modules and libraries, shared within the community.

9.2 Libraries

One thing that we missed very often during theory development are ready-to-use theory libraries to help solve the problems at hand. It turns out that the design and development of such libraries is both difficult and critical. It is difficult because the libraries' usefulness strongly depends on tiny formalization details. It is critical because fixing early design errors can result in costly adaptations in the theories of the (hopefully) many users of the library. Providing a library comes with the commitment to maintain and extend the library to prevent bit-rot and make the library more attractive to its users. Ideally, a librarian should be appointed for each library, who is responsible for development and integration of new results into the library (as opposed to a community-based approach of maintaining libraries, which in our experience does not work well). Overall this means that dedicated resources should be allocated for library development, which is an often overlooked fact in the planning of projects.

9.3 Pervasive verification and automation

All the verification was conducted in Isabelle/HOL (Version 2005). The main source of automation we employed were Isabelle's built-in tactics, like term rewriting (simp), the Tableaux-based prover for propositional logic (blast), the arithmetic decision procedure for Presburger arithmetic (arith), and combinations of those. One general observation is that the performance of those tools drastically suffers from large formulas that occur when dealing with lots of assumptions and invariants. This problem can be effectively avoided when preferring structured Isar proofs over unstructured apply-style proofs, since in Isar proofs the reasoning steps happen to be applied in smaller local context with only the relevant assumptions mentioned in the proof text. However, developing proper Isar proof techniques means additional training of the users, who may like to stick to their familiar apply-style proofs they first learnt.

Additionally to the built-in Isabelle tactics, customized tactics were developed to a limited extent. A verification condition generator for the Simpl Hoare logic was developed and was routinely applied for the verification of C0 programs. Applying transfer theorems for concrete programs was supported using reflection to discharge validity constraints like typing of C0 statements. Moreover, a tactic was developed supporting the transfer from Simpl Hoare triples to the big-step semantics, cf. Section 3.4.1.

Furthermore, the Hoare logic was designed to allow the integration of software model checkers and automatic termination analysis [DMSS05]. An external tool can either be employed as a trusted oracle that only delivers its final outcome or as an untrusted one that has to deliver a proof that can be replayed within Isabelle. However, even in the first case the integration of external tools in the context of pervasive verification is quite challenging. As the results have to smoothly fit into the overall verification workflow, great care has to be taken to agree on the exact semantics of properties and every subtlety of the underlying model, e.g., the C0 language. The additional effort to tune a tool that can handle the task 'in principle' to one that exactly fits into its scope is easily underestimated and scientifically often not rewarding. We consider this the main reason that for software verification the integration got stuck in an experimental phase. However, even in case of a successful integration the percentage of the proof obligations for the complete functional verification of C0 programs we expected to be able to discharge was only around 10 to 15 percent. As we were working in the context of interactive theorem proving the formalization is biased towards concise intelligible specifications, which exploit the expressibility of HOL. When focusing on automation it may be beneficial to consequently restrict oneself to fragments like first order logic. However, this comes at the cost of readability.

After all we see two promising sources to improve automation. First of all experienced users that make the 'right' formalizations, and second the design of libraries and the smooth integration of small decision procedures (e.g., handling bit vectors) as well as general purpose provers (like automated first order or higher order provers) that help discharging routine proof obligations.

9.4 Best practices

In contrast to theory repositories like the *archive of formal proofs*,[11] where the contributions mostly stand only on their own, all theories developed for the pervasive

[11] http://afp.sf.net

verification of a system must in the end fit together and be checked simultaneously. The required level of integration in pervasive verification is therefore much higher than with conventional verification and some difficulties mirror those in the development of large software projects. Hence, it is not surprising that many useful practices can be absorbed from the software engineering field.

We have broken down overall (theory) development into a number of modules with acyclic dependencies among each other. Modules are stored in a version control system; branches are used to maintain variants of modules (e.g., depending on different Isabelle versions). While the prover could in principle live with all definitions and proofs stuffed into a single theory, module and theory structure are intended to simplify human understanding of the theory corpus.

We use a (semi-automatic) continuous integration approach to check all theories regularly in a clean build environment. All theories are processed in a single heap. This strategy of early merging has helped us find problems with the prover and with the theories. Problems in the first category were for example heap size limitations, which were overcome with newer Poly/ML versions. Problems in the second category were in the beginning for example conflicting simpset modifications in theories or tweaking of global configurations options in (old versions of) Isabelle (e.g., modifying the simplification depth limit), which are very hard to locate and are therefore best detected early. Once the initial merging phase was completed, the typical problem in the second category of course was the breakage of higher-level theories due to changes in lower-level theories. While we used regular heaps for running the integration, we did not make use of Isabelle's session mechanism, which was too inflexible for theory checking in a fast-changing environment: Isabelle sessions are organized in a linear fashion (not in a DAG), therefore cannot be merged, and with sessions theories can only be checked at a session granularity, regardless of whether only a few theories need to be rerun. Instead, we have used an open, live heap for theory checking, such as is used when running an interactive session. In this scenario, if a theory changes, 'only' the theory itself and all theories that import it need to be checked.[12]

9.5 Some notes on the formalization

We summarize some of our experiences with the different formalizations of the C0 semantics. In our formalizations we did not distinguish between real runtime faults, e.g., caused by array index violations or division by zero that we want to reason about, and malformed situations that are ruled out by system invariants like type safety. With a transition *function* like δ within a logic of total functions one has to define some behavior even for those malformed configurations that we are not really interested in. If we had used a small-step *relation* instead one could focus only on the good configurations and otherwise just get stuck. A type safety theorem would then guarantee progress for valid configurations. The functional definition has the drawback that one always has to go through case distinctions to rule out those malformed configurations during the proofs about the semantics; there are many of those proofs in our project.

The big-step semantics has a quite strict separation between type information and the memory, whereas a memory configuration in the small-step semantics maintains

[12] As we will argue later on, this is still too slow to provide a pleasant user experience for certain tasks.

both. Similar decisions are sometimes taken within the definitions of auxiliary functions and predicates. For example there is a certain trade-off between defining a single function that recurses over the syntax tree once and calculates multiple results simultaneously, or to define several similar functions for the different results. Our experience is that keeping things separate is beneficial in the long run, in particular when formal entities may be used in different scenarios that one has not thought of in the beginning. For example one designs something like typing with the compiler correctness proof in mind, and later on similar things are needed in simulation or soundness proofs. In the worst case one duplicates work and develops a similar notion for the new application. Having smaller formal entities supports simpler theorems that are likely to be more general as a combined theorem for a more complex definition. This makes the formalization more flexible in the end.

9.6 TP system improvements

We mention some improvements to the theorem-proving system (rather than the logics, models, or proofs), which would have helped in developing and enhancing the quality of our theory corpus. Notably, most of the suggested improvements are applicable in two contexts: online, during theory developing to improve the user interface and experience, and offline, during theory checking, benchmarking, or analysis of other kind.

Speed. With a large theory corpus such as ours, improving TP speed would be a big gainer. Improving the speed comes in two varieties: reducing turn-around times and reducing the latency for (full or incremental) theory checking. The latter improvement is relevant in interactive context only.

When people work interactively with their own theories, they often enable Isabelle's *skip proofs mode*, which boils down to Isabelle just parsing and type-checking all imported theories. Although for higher-level theories, it can take on the order of an hour to process all imported theories, this usually reduces the time required to load the theories that you are not working on to a bearable minimum. However, this is only a work-around and this behavior is unsafe: from time to time broken proofs of lemmas in the current theory working set will not be noticed due to having been accidentally skipped altogether.

If lower-level theories are modified (e.g., because of adding a new lemma to a theory library or moving an existing lemma to a more appropriate theory) good practice dictates that theories are checked from the modified theory upwards. This operation cannot be done when skipping proofs, and even checking up to your own theories only (omitting other theory branches reaching up from the modified theory) can result in unbearable delay. This has discouraged people from adding lemmas at the right place in the first place or cleaning-up their theories after they have finished their development.

There has been already much progress with recent Poly/ML versions, which support multi-threading. Recent Isabelle versions make use of this parallelism, by decoupling proof checking from theory loading and checking independent proofs in parallel. The new system offers great flexibility in scheduling which proofs to check when, which can lead to a significant reduction of latency when working interactively with the prover.

Apart from taking advantage of multiple cores at the theory or proof level, the turn-around times in incremental proof checking will not be reduced by this approach.

Reducing the turn-around times for changes in low-level theories, which is in the order of hours, by a small factor only (i.e., the number of cores) will not be sufficient. We think that significant reductions in turn-around time can be achieved by tracking dependencies of proofs on a more fine-grained basis and then basically checking a proof only if its dependencies have changed. For example, if in a change only the proof of single lemma has changed but not its claim then only this single proof has to be checked to get an up-to-date heap. In principle, Isabelle can keep track of dependencies (in newer version it does so by default). Mechanisms for change tracking, however, are not yet present in Isabelle and would have to be implemented.

A wholly different kind of approach aims at reducing the accumulated 'global' turn-around time, i.e., the CPU cycles a group of developers burn cooperatively working on common goal, e.g., proving a computer system correct. This duplication of effort can be reduced by establishing a (possibly decentralized) heap / theory / proof distribution network or working with a server-based approach ('proof wikis'). A server-based approach can leverage existing continuous integration infrastructure. However, theory developers as much as software developers do not work in one 'universe' all the time, but rather in multiverses, in which branching and merging occurs when the development of new features begins and ends. This has to be taken into account in the design of a server-based solution.

Tools for end-users. Similar to powerful integrated development environments (IDEs), TP systems should provide more tools for end-users, to enable the analysis and improvement of the quality of the theories that are developed. In a non-interactive setting, such tools should enable to continuously control theory quality and warn about new hot-spots in the theory corpus. Naturally, the level of 'understanding' of these tools will have to stay quite low. There are still a couple of interesting areas where relatively simple tools can help.

First, there is the area of exploration and navigation of the theory corpus. This helps in reading, understanding, and most importantly using other people's theories and theorems. Thus, exploratory tools should help prevent duplicate effort in theory development; in a non-perfect world, you may find useful, proven results at the place where you least expect them. Isabelle's find-theorems facility is a nice example of such a tool: it can find theorems via higher-order pattern matching. It will however not find theorems outside of the current theory branch, causing users to resort to inferior tools like `grep` instead. This problem of focus can be avoided by searching for theorems on big heaps only, which include the whole theory corpus.

Second, there is the area of slenderizing and massaging the theory corpus. There should be tools that give hints online during theory development ('this goal has already been proven') or afterwards ('this lemma can be moved elsewhere'). Also, the identification of dead parts of the theory corpus is of interest. These tools are required because theory developers often work locally first, distributing their results only when development has finished (partly, this is done for speed reasons, cf. above). We experimented with five approaches in this area:

1. We applied ConQAT, a clone detection tool,[13] to our theory sources. A first analysis shows that up to 28% percent of the text in typical theories is redundant. Because proof structure is generally ignored and sometimes duplicate proofs are actually valid, this very simple first analysis can produce false positives. A quick inspection of

[13] http://conqat.in.tum.de/

the reported clones however showed that a good fraction really could be factored out into separate lemmas.

2. Similarly, we have implemented a tool to detect recurring conclusions of subgoals (across many proofs). In an online analysis, the prover can warn you if you are about to establish a conclusion that you have already proven earlier. In an offline analysis, recurring conclusions of subgoals can just be reported by number of recurrences; the higher the number of repetitions, the higher the potential merit in factoring out the conclusion into a separate lemma. In this case, the prover can also suggest potential assumptions for the lemma, e.g., the intersection of the sets of assumptions that have appeared for that particular conclusion. An initial analysis for the C0 small-steps semantics (which contains many lemmas, e.g., on type safety) has shown that ca. eight percent of the subgoal conclusions appear repeatedly. This number should be treated with care, however, because if a certain chain of conclusions appears multiple times (because the same proof is conducted multiple times), than all conclusions in all instances of the chain are counted multiple times. With apply-style proofs, however, these chains cannot easily be filtered out, because in contrast to Isar-style proofs, there is no concept of a 'sub proof'.

3. With lack of proper exploratory tools and lemmas sometimes being placed at odd locations, some results may have been proven multiple times. This has happened a couple of dozen times in our theory corpus. In most cases, only auxiliary results and simple lemmas were duplicated. We however also had cases, where larger lemmas became identical due to refactoring (e.g., removing a model parameter).

Lemma clone detection checks for multiple lemmas having the same claim. We have performed experiments, checking for lemma duplicates both at the source level and inside Isabelle, looking for almost identical clones only. This analysis can be improved by performing more relaxed comparison of lemma claims; e.g., generalizing over bound and unbound variables or doing more aggressive normalization. Another possibility would be to try using ATPs (e.g., integrated in newer Isabelle versions via the 'sledgehammer' tool) to detect clones or nearly-clones of lemmas.

4. To help placing lemmas in theories where they might best belong, we have developed a tool called 'Gravity'. As noted several times earlier, during theory development (auxiliary) lemmas are often placed in higher-level theories for convenience. By analyzing the lemma's dependencies, the Gravity tool can compute (minimal) theories[14] that satisfy the dependencies needed by the lemma. Performing the analysis for a whole group of lemmas (e.g., all lemmas in a theory), a better result can be produced by 'skipping' intra-group dependencies. This way the tool would suggest moving two lemmas that are in the same theory and where second lemma uses the first lemma to the same minimal theories.

5. Finally, it can be important to detect and prune dead parts of the theory corpus. These may exist for historical reasons or due to refactoring operations as described above. Excessive pruning may be harmful, though. Depending on the analysis being employed and the scope at which it is operating, exported theorems of a library or even top-level theorems may be classified as unused; clearly, these should not be removed. We have written a simple prototype to conservatively detect unused lemmas at the source code level; we have manually removed a couple of dozens lemmas, which were reported by this tool.

[14] A unique minimal theory may not exist because theories dependencies form a DAG.

Finally, there is the area of benchmarking. In continuous integration, benchmarking information is vital for time-out detection – checking a theory corpus must not be delayed indefinitely because, e.g., a new, unfortunate simplification rule causes the simplifier to loop in just a single lemma of the whole corpus. Also, continuous benchmarking in interactive use helps raise awareness on bottlenecks in proofs, which may point out to real performance problems in the formalization but also the prover itself.

A Glossary

C0 General

s, e, v	C0 statement, expression, and value
T	C0 type
x	extended state component for XCalls
$\lceil s \rceil$	decomposition of a statement
$\lceil\!\lceil s \rceil\!\rceil$	decomposition of a statement and removal of SKIP statements
tt	type table
pt	procedure table

C0 Big-Step

Π	C0 program
σ, τ	state in Simpl and C0 big-step
ext	extended state component
heap	heap component of state
lcls	local variables of state
glbs	global variables of state
free-heap	free heap locations of state
HT, GT, LT	heap-, global-, local typing
$\Pi,sz,L\vdash_{\mathsf{bs}} \langle s, \sigma\rangle \Rightarrow \tau$	big-step execution of a C0 statement
tnenv	type table of a program
genv	global variables of a program
plookup	procedure lookup in a program
xplookup	lookup of extended procedure in a program
$xpt,pt,tt,VT,HT\vdash_{\mathsf{bs}} s\ \surd$	typing of C0 statement
$HT,tt\vdash_{\mathsf{bs}} v{::}_{\mathsf{v}}T$	typing of value
$HT,tt\vdash_{\mathsf{bs}} m{::}MT$	typing of memory
$tt\vdash_{\mathsf{bs}} \sigma{::}HT,LT,GT$	typing of state
valid-prog	valid C0 program
$\mathcal{A}$	set of definitely assigned variables
$\mathcal{D}$	definite assignment analysis
$\Gamma\vdash_{\mathsf{h}} \langle s, \sigma\rangle \Rightarrow \tau$	big-step execution of a Simpl statement
$\Gamma\vdash_{\mathsf{h}} s{\downarrow}\sigma$	guaranteed termination of a Simpl statement
$\Gamma\models_{\mathsf{h}} P\ s\ Q$	total correctness of a Simpl statement

C0 Small-Step

c	configuration
mem	memory configuration
prog	program rest
gm	global memory
hm	heap memory
lm	local memory stack (frame stack)
ct	memory content
st	symbol table
init-vars	set of initialized variables
g-variable	variables and sub variables
$tt,pt,enough\text{-}heap,xpt\vdash_{\mathsf{ss}} c \rightarrow c'$	small-step relation for C0 statements

$\delta\ tt\ pt\ enough\text{-}heap\ c$	small-step transition function
$\delta^n\ tt\ pt\ enough\text{-}heap\ c$	small-step transition function (many steps)
$\delta_{\mathsf{x}}\ tt\ pt\ enough\text{-}heap\ c\ x\ xpt$	small-step transition function with XCalls
$\delta^n_{\mathsf{x}}\ tt\ p\ enough\text{-}heap\ c\ x\ xpt$	small-step transition function with XCalls (many steps)
$\mathsf{valid\text{-}C0SS}$	valid configurations
$\mathsf{valid\text{-}cfg_{ss}}$	valid configuration for property transfer
$tt,pt,enough\text{-}heap,xpt,valid_{\mathsf{asm}},L,\Pi \models_{\mathsf{ss}} P\ s\ Q$	total correctness
$\mathsf{transition\text{-}invariant}$	transition invariant

Compiler

$alloc$	allocation function from compiler theorem

VAMP

n, N	Step numbers for systems without and with devices
asm	VAMP assembly configuration
$\delta_{\mathsf{asm}}\ asm$	VAMP assembly transition function
$\delta^n_{\mathsf{asm}}\ asm$	VAMP assembly transition function (many steps)
isa	VAMP instruction-set architecture configuration
$valid_{\mathsf{asm}}$	Well typedness of assembly instructions
$\delta_{\mathsf{isa}}\ isa\ eev\ mifo$	VAMP ISA transition function
$devs$	Devices configuration
$\delta_{\mathsf{dev}}\ eifi\ mifi\ d$	Device transition function
$\delta^*_{\mathsf{dev}}\ eifis\ mifis\ d$	Device transition function (many steps)
$\mathsf{intr}\ d$	Device interrupt predicate
seq_{asm}	Execution sequence for VAMP assembly with devices
asm_{D}	Configuration for VAMP assembly with devices
$\Delta^N_{\mathsf{asm}}\ seq_{\mathsf{asm}}\ asm_{\mathsf{D}}$	VAMP assembly with devices transition function
seq_{isa}	Execution sequence for VAMP ISA with devices
$eifis$	Sequence of external inputs
isa_{D}	Configuration for VAMP ISA with devices
$\Delta^N_{\mathsf{isa}}\ seq_{\mathsf{isa}}\ eifis\ isa_{\mathsf{D}}$	VAMP ISA with devices transition function

CVM, Page-Fault Handler

Π_{k}	(abstract) kernel
pid	process identifier
$\Delta_{\mathsf{cvm}}\ cvm\ seqx_{\mathsf{asm}}$	CVM transition function
$\Delta^N_{\mathsf{cvm}}\ seq_{\mathsf{asm}}\ cvm$	CVM transition function (many steps)
x_{pfh}	extended state component for the page-fault handler
xpt	extended procedure table (for XCalls)
pfh	(abstract) page-fault handler configuration
pd	(abstract) page descriptor configuration
pcb	(abstract) process control block

References

[ABP09] Eyad Alkassar, Sebastian Bogan, and Wolfang Paul. Proving the correctness of client/server software. *Sādhanā: Academy Proceedings in Engineering Sciences*, 34, 2009. To appear.

[AH08] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In Shankar and Woodcock [SW08], pages 225–239.

[AHK$^+$07] Eyad Alkassar, Mark Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. Formal device and programming model for a serial interface. In Bernhard Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, pages 4–20. CEUR-WS.org, 2007.

[AHL$^+$08] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Shankar and Woodcock [SW08], pages 209–224.

[ASS08] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In C. R. Ramakrishnan and Jakob Rehof, editors, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS08)*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[Bal03] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 – May 4, 2003, Revised Selected Papers*, volume 3085 of *LNCS*, pages 34–50. Springer, 2003.

[Bal06] Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK, August 11–12, 2006, Proceedings*, volume 4108 of *LNCS*, pages 31–43. Springer, 2006.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An approach to systems verification. *JAR*, 5(4):411–428, December 1989.

[BHW06] Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification – Experiences from the Verisoft email client. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006)*, volume 192 of *CEUR Workshop Proceedings*, pages 112–125. CEUR-WS.org, August 2006.

[BJK$^+$03] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In Daniel Geist and Enrico Tronci, editors, *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BJK$^+$06] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006.

[Bog08] Sebastian Bogan. *Formal Specification of a Simple Operating System.* PhD thesis, Saarland University, Computer Science Department, August 2008.

[Bur72] R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.

[Con06] Cosmin Condea. Design and implementation of a page fault handler in C0. Master's thesis, Saarland University, July 2006.

[DDB08] Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings, 5th International Verification Workshop (VERIFY), Sydney, Australia*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, August 2008.

[DDW09] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *JAR: Special Issue on Operating Systems Verification*, 2009. To appear.

[DDWS08] Matthias Daum, Jan Dörrenbächer, Burkhart Wolff, and Mareike Schmidt. A verification approach for system-level concurrent programs. In Shankar and Woodcock [SW08], pages 161–176.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In Dominique Borrione and Wolfgang Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.

[DMSS05] Matthias Daum, Stefan Maus, Norbert Schirmer, and M. Nassim Seghir. Integration of a software model checker into Isabelle. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2–6, 2005, Proceedings*, volume 3835 of *LNCS*, pages 381–395. Springer, 2005.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.

[HEK$^+$07] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 41(4):3–11, 2007.

[HKS08] Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors. *3rd intl Workshop on Systems Software Verification (SSV 2008)*, volume 217C of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 2008.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HP08] Mark A. Hillebrand and Wolfgang Paul. On the architecture of system verification environments. In Karen Yorav, editor, *Hardware and Software, Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23–25, 2007*, volume 4899 of *LNCS*, pages 153–168. Springer, 2008.

[IT08] Tom In der Rieden and Alexandra Tsyban. CVM – A verified framework for microkernel programmers. In Huuck et al. [HKS08].

[Kle99] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

[Kle09] Gerwin Klein. Operating system verification – An overview. *Sādhanā: Academy Proceedings in Engineering Sciences*, 34, 2009. To appear.

[Lei08] Dirk C. Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Department, July 2008.

[LNRS07] Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan. Verification of distributed applications. In Francesca Saglietti and Norbert Oster, editors, *Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18–21, 2007*, volume 4680 of *LNCS*, pages 315–328. Springer, 2007.

[LP08] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification – From verified programs to verified systems. In Huuck et al. [HKS08], pages 23–40.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NYS07] Zhaozong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *TPHOLs '07*, pages 189–206. LNCS, 2007.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, May 2007.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, volume 3452 of *LNCS*, pages 398–414. Springer, 2005.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich, April 2006.

[ST08a] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In Huuck et al. [HKS08], pages 169–185.

[ST08b] Artem Starostin and Alexandra Tsyban. Verified process-context switch for C-programmed kernels. In Shankar and Woodcock [SW08], pages 240–254.

[SW08] Natarajan Shankar and Jim Woodcock, editors. *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *LNCS*, Toronto, Canada, October 2008. Springer.

[TS08] Sergey Tverdyshev and Andrey Shadrin. Formal verification of gate-level computer systems. In Kristin Yvonne Rozier, editor, *LFM 2008: Sixth NASA Langley Formal Methods Workshop*, NASA Scientific and Technical Information (STI), pages 56–58. NASA, 2008.