The Verisoft Approach to Systems Verification

Eyad Alkassar[1,⋆], Mark A. Hillebrand[2,⋆⋆], Dirk Leinenbach[2,⋆,⋆⋆], Norbert W. Schirmer[2,⋆⋆], and Artem Starostin[1,⋆⋆⋆]

[1] Universität des Saarlandes
P.O. Box 15 11 50, 66041 Saarbrücken, Germany
{eyad,starostin}@wjpserver.cs.uni-sb.de

[2] German Research Center for Artificial Intelligence (DFKI)
P.O. Box 15 11 50, 66041 Saarbrücken, Germany
{mah,dirk.leinenbach,norbert.schirmer}@dfki.de

Abstract. The Verisoft project aims at the pervasive formal verification from the application layer over the system level software, comprising a microkernel and a compiler, down to the hardware. The different layers of the system give rise to various abstraction levels to conduct the reasoning steps efficiently. The lower the abstraction level the more details and invariants are necessary to ensure overall system correctness. Illustrated by a page-fault handler we discuss the layers and the trade-off between efficiency of reasoning at a more abstract layer versus the development of meta-theory to transfer the verification results between the layers.

1 Motivation and Challenges

The layer of system software confines the essential components of modern computer architectures. Any flaw up to this level has a decisive impact on the robustness, safety, and security of applications running on top of it. An operating system kernel that might fail to guarantee isolation of processes can hardly serve as a trustworthy computing basis to process security critical data. Hence, the design and verification of the crucial system level parts by the most rigorous means is an effort both worthwhile and promising.

Examining the system design up to the level of a microkernel, we typically have to deal with at least the following layers: hardware, assembler, and the C programming language. The different layers come along with a rise in abstraction regarding formal models and reasoning about them. However, the final goal is to provide objective evidence that the actual running system behaves correctly. The lower the layer the 'correctness theorem' holds on the better. Ideally,

⋆ Work was supported by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems'.

⋆⋆ Work was supported by the German Federal Ministry of Education and Research (BMBF) within the Verisoft project under grant 01 IS C38.

⋆⋆⋆ Work was supported by the International Max Planck Research School for Computer Science (IMPRS-CS).

Appeared in:
N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp.209-224, 2008

this is a theorem in the domain of physics. For computer science a transistor or gate-level hardware model is a realistic target to state the final correctness result. Employing higher abstraction levels to improve effectiveness of reasoning demands that we close the 'semantic gap' and bring the results down to the hardware level. This is the very idea of *pervasive* or *systems* verification [1,2]. In Verisoft every abstraction layer is justified by meta-theorems that allow transferring the results to the low-level models. All the development is mechanized in the uniform logical framework of the interactive theorem prover Isabelle/HOL and hence it is rigorously checked that all the results fit together.[3] The goal of this paper is to provide an informal overview of the different layers and their connection. This bird's eye view easily gets lost in detailed technical papers on parts of this work that were already or are simultaneously published.

Related Work. First attempts to use theorem provers to specify and even prove correct operating systems were made as early as the 1970ies in PSOS [3] and UCLA Secure Unix [4]. However, a missing or to a large extent underdeveloped tool environment made mechanized verification futile. With the CLI stack [1], a new pioneering approach for pervasive systems verification was undertaken. The goal of this project was to build a system from verified, hierarchically stacked components. In extension to their seminal work the Verisoft project aims at a more realistic system architecture regarding both hardware and system software. In particular, devices are integrated into the Verisoft system stack. For realistic systems, this is already required for booting or scheduling in a microkernel. It is theoretically challenging, since devices are a concurrent source of computation and break the abstraction of sequential programs.

The project L4.verified [5] focuses on the verification of an efficient microkernel, rather than on formal pervasiveness, as no compiler correctness or an accurate device interaction is considered. The microkernel is implemented in a larger subset of C as C0 (the C-like programming language used in Verisoft), including pointer arithmetic and an explicit low-level memory model [6]. However, with inline assembler code we gain an even more expressive semantics as machine registers become visible if necessary. So far, only exemplary portions of kernel code were reported to be verified, the virtual memory subsystem uses no demand paging [7]. For code verification L4.verified relies on Verisoft's Hoare environment [8]. In the FLINT project, an assembly code verification framework is developed and code for context switching on a x86 architecture was formally proven [9]. A program logic for assembler code is presented, but no integration of results into high-level programming languages is undertaken.

The VFiasco project [10] aims at the verification of the microkernel Fiasco implemented in a subset of C++ and embedded into PVS. There is no attempt to map the results to the machine level.

Overview. In Sect. 2 we give an overview of the Verisoft system stack and our approach towards pervasive verification. We proceed in Sect. 3 by introducing the C0 language stack from a Hoare logic down to a low-level small-step C0

[3] Theory files are available at http://www.verisoft.de/VerisoftRepository.html.

semantics. The compiler verification detailed in Sect. 4 is the bridge between C0 and the machine model. In Sect. 5 we explain how machine-level entities can be made accessible from within the high-level Hoare logic based reasoning and Sect. 6 explains the integration of devices. In Sect. 7 we illustrate the approach on the example of a page-fault handler implementing demand paging for memory virtualization. We conclude in Sect. 8.

2 Pervasiveness

In short, pervasive verification means that at the end of the day we obtain a correctness theorem at the lowest level of abstraction, the machine level. To avoid conducting all the verification at the machine level in the first place we introduce layers (like a C0 semantics and a Hoare logic) to improve the level of abstraction along with the performance of verification. Nevertheless, meta-theoretic theorems allow us to bring the verification results all the way down to the machine level, where they can be composed for an overall system correctness proof. Figure 1 depicts our system stack. The bottom layer is given by the gate-level description of the VAMP hardware, our hardware platform. A processor correctness result [11] links this to the layer of the instruction set architecture (ISA), where instructions and values are encoded as bit-vectors. The VAMP assembler layer formalizes the programmer's view on the machine and is a slight abstraction of the ISA layer, where the bit-vector encodings of the instructions and values are switched to an abstract data type and natural numbers, respectively. A straightforward simulation theorem connects these layers. Assembler computations are described by a small-step semantics. The high-level programming language C0 is also formalized with a small-step semantics and a compiler correctness result relates C0 computations to assembler computations. This is the junction where inline assembler code and concurrent device interaction can be merged into purely sequential C0 code. At the upper levels the inline assembler code is abstracted to so-called XCalls. A big-step semantics serves as intermediate layer towards a Hoare logic, which is our main vehicle for reasoning about C0 programs. Soundness of the Hoare logic and simulation between the operational semantics allow us to transfer the Hoare triples down to the small-step semantics.

3 C0 Semantic Stack

The C0 programming language is a subset of C, designed to be expressive enough to allow the implementation of large parts of the system software, while remaining 'neat' to verify by restricting ourselves to a type-safe fragment without pointer arithmetic. In the lucky position not having to verify already existing C system code we could develop the code base ourselves in parallel with the specification and verification. Hence, it was reasonable to restrict ourselves to a subset of C to make the code verification less tedious and detailed, and thus make it faster. However, it is intrinsic to system level code to perform low-level

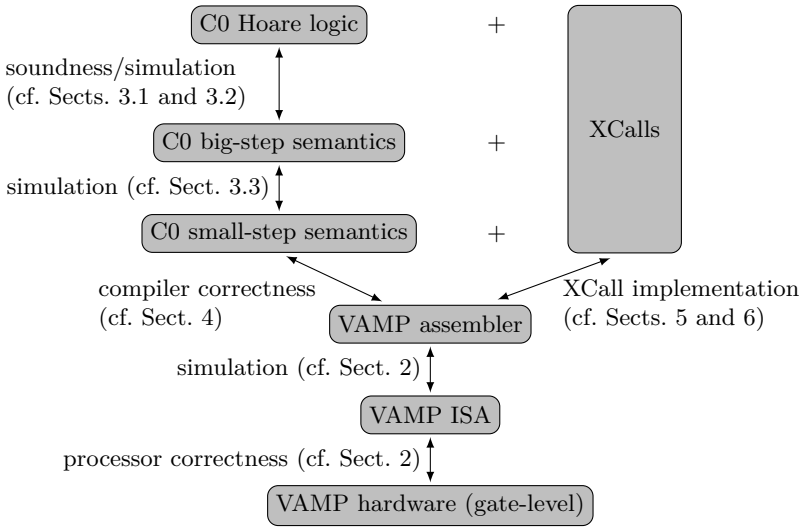

Fig. 1. System Stack

architecture dependent operations. We clearly cannot avoid those parts without sacrificing any correspondence to real systems. Our approach is to encapsulate those tricky low-level parts in inline assembler code. We have to deal with approximately 99% of type safe C0 code and 1% of assembler.

The C0 semantics plays a central role. On the one hand the Hoare logic has to be sound with respect to the C0 semantics, on the other hand it is a cornerstone of the compiler correctness theorem that links the formal C0 semantics to the machine semantics of the generated code. All the Hoare logic based code verification, as well as the C0 semantics, the compiler and the meta-theorems are mechanized in the theorem prover Isabelle/HOL. It is challenging to come up with a formalization that is equally well equipped to support both the verification of individual programs besides the meta-theory. For example, to specify and verify a C0 compiler it is natural to explicitly represent C0 types, such that the compiler can exploit this information. To reason about an individual C0 program however, this type information is not explicitly necessary or can be even obstructive. In C0 there is neither a means nor a need to access type information from within the program. Quite the opposite, the C0 programs we consider are all well-typed and C0 is a type safe language. Hence it is tempting to find a way to abstract from potential typing issues while reasoning about individual programs. In a typed logical framework like HOL this can be achieved by mapping programming language types for C0 variables directly to HOL types. The built in type inference of Isabelle/HOL then automatically takes care of typing. While this is a reasonable approach for a Hoare logic, it is not well-suited as a formalization of the C0 semantics itself and the accompanying meta-theory. In this context an explicit representation of programming language typing is necessary

in order to even express things like 'type-safety'. These conflicting requirements on 'the' C0 semantics (together with further ones described below) is the reason we defined a complete C0 semantics stack. To preserve our goal of pervasive verification the semantics are linked together via meta-theorems.

A more fundamental difference in the semantic formalization is the decision between big-step and small-step semantics. C0 is a sequential language and even the target machine is a uni-processor architecture. Sequential reasoning and a big-step semantics seem to be adequate. However, the processor is not the only relevant hardware unit at the level of system software: devices run concurrently with the processor, and communication involves interrupt handling and memory-mapped I/O. This concurrency makes a small-step semantics adequate. To cover the sequential and concurrent aspects we introduce both a big-step and a small-step semantics.

The big-step semantics serves as a target for Hoare logic based reasoning: a Hoare triple is a 'big-step' program property relating the initial and final state. In general, a small-step semantics is more expressive than the big-step counterpart. It describes the complete trace of the computation rather then just the initial and final state. Moreover, it can be used to reason about non-terminating computations, whereas the big-step semantics does not distinguish between stuck computations and non-termination. A big-step semantics however is more abstract (e.g., an explicit frame-stack for procedures is unnecessary) and supplies a powerful proof principle: rule induction (i.e., induction on the depth of the derivation tree). The similar structure of both a big-step semantics and a Hoare calculus makes its soundness proof quite straightforward.

An orthogonal issue is the representation of programming language values. C0 offers aggregate values as arrays, structures, or combinations of both. For reasoning about C0 programs it is both sufficient and comfortable to keep this abstraction of aggregate values: a single 'memory cell' can contain an aggregate value of arbitrary size. The real memory of the target architecture however is byte addressable and aggregate values are stored in a consecutive sequence of bytes. We flatten aggregate values to byte sequences in the small-step semantics whereas we treat them as compact atomic entities in the big-step semantics.

Another aspect of aggregate values is their representation on the heap, in particular considering dynamic pointer structures. Aliasing is one of the key obstacles when reasoning about pointer programs. Any reduction of potential aliasing means a significant benefit for verification. Type-safety of C0 prohibits aliasing between pointers to different types. We exploit this invariant by switching to a different memory model in the Hoare logic. Instead of a single monolithic heap that stores every kind of value, we introduce a separate heap [12] for each field of a structure. Without further reasoning the model already rules out aliasing between different types or even different structure fields.

3.1 Simpl Hoare Logic

The Hoare logic environment [8], built on top of the Isabelle/HOL, was motivated by C0 but is by no means restricted to C0. It is a self-contained theory

development for a quite generic model of a sequential imperative programming language called *Simpl*. A big-step semantics as well as a Hoare logic for both partial and total correctness is defined. Soundness and completeness is proven.

Theorem 1 (Soundness). *Every triple derived in the Hoare logic is valid with respect to the operational semantics.*

Theorem 2 (Completeness). *Every triple that is valid with respect to the operational semantics can be derived in the Hoare logic.*

Soundness is crucial for pervasive verification in order to formally link the results from the Hoare logic to the operational semantics. Completeness can be viewed as a sophisticated sanity check for the Hoare logic, ensuring that one cannot get stuck in the verification because of some missing Hoare rules.

The state-space representation within Simpl is not fixed, rather it is a HOL type variable. It can be instantiated to meet the requirements and format of the program one attempts to verify. In Simpl all expressions are shallowly embedded, whereas compound statements are deeply embedded. The basic atomic statements however, are also modeled semantically as a state update function. This careful arrangement makes it possible to define and reason about the Hoare logic via the statement structure of the program and additionally provides the flexibility to instantiate the language with various state-space models or atomic operations that reflect the programming language under consideration.

To facilitate the usage of the Hoare logic within Isabelle/HOL, the application of the rules is automated as a verification condition generator. Moreover, an interface to software model checkers and termination analysis is provided [13].

3.2 Simpl to C0 Big-Step

We embed C0 into Simpl and use its verification environment to conduct C0 program proofs. To be able to transfer Simpl Hoare triples down to C0, we make sure that every behavior of the C0 program is also part of the Simpl counterpart and is thus covered by the Hoare triple. The C0 representation in HOL is a traditional deep embedding. First the abstract syntax of all relevant entities—types, values, expressions, and statements—is defined and then meaning is assigned to these entities, via a type system or the operational semantics. On the C0 syntax we define an abstraction to Simpl programs. Similarly, we define an abstraction of the C0 program state to the corresponding Simpl state. The following key theorems allow us to transfer Hoare triples from Simpl to C0.

Theorem 3 (Simpl simulates C0). *Every execution of a C0 program is simulated by the execution of the corresponding Simpl program.*

Theorem 4 (Preservation of termination). *Termination of the Simpl program implies termination of the original C0 program.*

The proof of Theorem 3 is conducted by induction on the big-step execution of the C0 program (cf. [8, Chapter 8]). The most important invariant used for the induction is type safety. This matches the intuition of the abstraction that goes on between the C0 and the Simpl representation. In C0 typing is explicit whereas in Simpl it is mapped to HOL typing. To be more accurate Theorems 3 and 4 only hold for well-typed C0 programs and well-typed memory states. In C0 those constraints are encoded in predicates whereas in Simpl they are maintained by Isabelle's type-inference. A type-safe memory is also the prerequisite to justify the split heap representation in Simpl against the monolithic heap in C0.

A technical issue of the simulation proof comes from the state-space representation in Simpl. As we use a HOL record to encode program variables and heap components the HOL type of the state is not fixed for all C0 programs, but depends on the individual program. We achieve an individual HOL typing for the different variables and heap components, but cannot formulate Theorems 3 and 4 generically for all C0 programs within HOL. Instead we come up with a stratification of the theorems in two stages. A meta-theorem holds for all C0 programs and *assumes* commutation properties for the atomic state lookups and updates that appear in the program. For each individual C0 program we separately *discharge* these commutation properties. Fortunately, the second step is straightforward and can be automated. Hence, the resulting methodology is still generically applicable for all C0 programs. Isabelle's lightweight module concept of locales [14] supports this separation into a meta-theorem for all programs as well as the instantiation [15] for individual programs.

3.3 From Big-Step to Small-Step

The theorems to transfer Hoare triples from the big-step semantics down to the small-step semantics are analogous to Theorems 3 and 4.

Theorem 5 (Big-step simulates small-step). *Every terminating C0 small-step computation can be simulated by a big-step execution.*

Theorem 6 (Preservation of termination). *Termination of the C0 program in big-step semantics implies termination in the small-step semantics.*

Intuitively, we have to bring two orthogonal aspects of the semantics together: (i) computation: from a sequence of steps to a big-step and (ii) memory: from flat to aggregate values. We reflect this separation in the proofs by introducing an intermediate semantic level. A small-step semantics with the same memory model as the big-step semantics as well as the same (computational) granularity as the small-step semantics. First we focus on the computation aspect and prove that a terminating sequence of computation steps in the intermediate semantics can be simulated by a single big-step execution and that termination of the big-step semantics implies termination of the intermediate small-step semantics. In the second step we focus on the memory and define an abstraction function that maps a small-step configuration to a configuration in the intermediate semantics.

We prove that every step in the small-step semantics can be simulated in the intermediate semantics, which also implies that termination is preserved.

Again, the simulation only holds for well-typed programs and well-typed memories. Since the memory representation varies in the different semantics the notion of a well-typed memory also differs. The low-level notion introduced in the small-step semantics together with the abstraction function to aggregate values is sufficient to imply the relevant restrictions at the big-step level.

4 Compiler Correctness

Software verification in Verisoft does not stop at the C0 level. To allow execution of verified programs on the 'real' hardware they are compiled to binary code. Of course, this translation could itself introduce errors into an otherwise verified C0 program. Thus, verification of the translation process is essential for pervasive systems verification if the system software and applications are implemented in a high-level programming language.

We close this gap by verifying a simple, non-optimizing compiler translating C0 to VAMP assembler code [16]. In addition to a *compiling specification* in Isabelle/HOL the compiler is *implemented* as a C0 program. Both the compiling specification and the compiler implementation have been formally verified. For the former we have proven a small-step simulation theorem stating that the original C0 program (executed by the C0 small-step semantics) and the compiled code behave equivalently. For the latter we have shown using our Hoare logic environment that it produces exactly the same list of assembler instructions as the compiling specification. Both results are combined into a single theorem. In this paper we can focus on the correctness of the compiling specification.

The correctness theorem presented below applies to *translatable programs*, which must be well-formed and fulfill certain resource restrictions of the target machine, e.g., to deal with limited memory size. Since the C0 small-step semantics and the VAMP assembler machine are deterministic this allows us to transfer Hoare triples down to the assembler and hardware layer.

The code generation algorithm of the C0 compiler follows directly the structure of the input program. It starts by iterating over all functions in the function table and generates code for their bodies. The code generation for statements and expressions—in the context of a certain function—is done by a simple recursive algorithm which follows the structure of the corresponding data types.

Essentially, the main theorem of the compiler correctness proof states that for all steps i of the C0 machine there exists a corresponding step number $s(i)$ such that after $s(i)$ steps the assembler machine is consistent with the C0 machine after i steps. This consistency is stated formally by a simulation relation between configurations of the C0 machine and configurations of the VAMP assembler machine. The simulation relation consists of several parts. *Control consistency* states that the VAMP's program counters point to the code of the first statement in the current C0 program rest. *Code consistency* requires that the compiled code of the C0 program remains intact, i.e., it forbids self-modification. *Value*

consistency requires for all *reachable* variables g of basic type that the (C0) value of g is stored in the VAMP configuration at the allocated address of g. For reachable pointer variables p which point to some variable g we require that the value stored at the allocated address of p in the VAMP machine is the allocated base address of g. This defines a subgraph isomorphism between the reachable portions of the heaps of the C0 machine and the VAMP machine. *Stack consistency* is a technical predicate about the implementation of the run time stack and the content of some special registers.

Because the set of valid variables of a C0 machine changes with *new* statements, function calls, and returns, and because garbage collectors can change the allocated base address of variables on the heap, the simulation relation is parametrized by the current allocation function, which maps variables to their allocated base address in the target machine.

Theorem 7 (Compiling Specification Correctness). *Let p be a translatable C0 program, and assume that the initial assembler configuration holds the compiled code of p. Then, for all steps i of the C0 machine executing program p that did not reach an error state or produced a stack overflow there exists an assembler step number $s(i)$ and an allocation function $alloc^i$ such that the C0 machine after i steps (of small-step semantics) is consistent with the assembler machine after $s(i)$ steps with respect to $alloc^i$.*

5 Extended Hoare Logic

The language stack described in Sects. 3 and 4 allows us to transfer program properties from the Hoare logic down to the assembler level. However, in the context of system code we have to deal with portions of inline assembler code that break the abstraction of structured C0 programs: low-level entities (like the state of a device) may become visible even in the specification of code that is only a client to the inline assembler parts. To avoid doing all the verification in the lower semantic levels we extend the Hoare logic to represent the low-level actions on an abstract extension of the state-space. Inline assembler code is encapsulated in so-called *XCalls* at the Hoare logic level, which are modeled as atomic state updates. The correctness proofs of Sect. 3 allow us to transfer XCalls down to the assembler semantics, where they are finally discharged by an implementation proof. The compiler correctness proof only covers the assembler free portions of the code. Assembler code is inserted literally into the target code. Since compiler correctness is stated relatively to small-step semantics (of both the C0 and the assembler machine) the results can be extended to the combined computation: As long as the ordinary C0 code is executed the compiler correctness theorem covers the computation of the translated code and guarantees that we arrive at corresponding configurations at the machine and the C0 semantics level. Then the inline assembler code is executed according to the assembler semantics. The implementation proof (which also includes termination) ensures that its effect is the one expected by the abstract XCall semantics. Hence, at the end of the

assembler computation we again arrive at corresponding final configurations. Then we can switch back to the compiler correctness result, and so on.

6 Dealing with Devices

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), need correctly implemented drivers. Hence, any verification approach of computer system stacks should deal with driver correctness. Nonetheless, when proving functional driver correctness it does not suffice to reason only about code running on a processor. Devices themselves and their interaction with the processor also have to be formalized.

We model devices as deterministic finite-state-machines communicating with both an external environment and the processor. The external environment is used to model non-determinism and communication; the network interface card, for example, sends and receives network packets. The processor accesses a device by reading or writing special address ranges. The devices, in turn, can signal interrupts to the processor; DMA is not considered. In Verisoft, we formalized models for an ATAPI disk, a serial interface, and an automotive bus controller.

At the gate-level hardware, devices are executed in lock-step. However, moving to the instruction set architecture, we loose granularity and hence timing information. We compensate for this loss by introducing interleaved execution of devices and the processor. An oracle, called *executing sequence*, determines when some device or the processor is allowed to take steps. In the following we refer to this interleaved semantics as the combined system.

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.

A basic observation of our overall model is that device and processor steps that do not interfere with each other can be swapped. For a processor and a device step, this is the case if the processor does not access the device and the device does not cause an interrupt. Similarly, we can swap steps of devices not communicating with each other. Utilizing this observation we reorder execution sequences into parts where the processor accesses no device or only one device. All interleaved and non-interfering device steps are moved to the end of the considered part and hence a (partially) sequential programming model is obtained.

Theorem 8 (Reordering of Non-Interfering Devices). *The interleaved execution of the combined system can be reordered into non-interleaved chunks of processor and device steps, such that the resulting execution simulates the original computation.*

Note that compiled, pure C0 programs never access devices, because data and code segments must not overlap with device addresses. Hence, all inter-

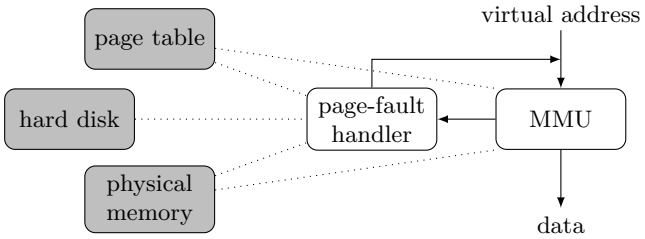

Fig. 2. Concept of Paging

leaved device steps can be delayed until some inline assembler statement is encountered. In combination with the concept of XCalls, lifting driver correctness statements to pure sequential Hoare logic, as demonstrated in the next section, becomes feasible. More generally, the execution of drivers controlling different (non-interfering) devices can also be separated, enabling modular verification of device drivers (the sketched reordering theory is developed and applied to a simple hard disk driver in [17]).

7 Property Transfer Example: Page-Fault Handler

The formal verification of an academic operating system microkernel is a Verisoft subproject. The microkernel contains a verified page-fault handler that, in collaboration with other memory management routines, implements isolated, virtual memory for the user processes by means of demand paging [18]. Implemented in about 300 lines of C0 code with several calls to a hard disk driver written in assembler, it is a perfect candidate to illustrate our verification methodology.

Problem. One of the most challenging parts of verification of the Verisoft microkernel is memory virtualization, i.e., to ensure that each user process controls its own, large, and isolated memory. User processes access memory by virtual addresses, which are subsequently translated to physical ones. Modern computer systems implement virtual memory by demand paging: small consecutive chunks of data, called pages, are either stored in a fast but small physical memory or in a large but slower swap memory (usually a hard disk). The page table, a data structure both accessed by the processor and by software, maintains whether a page is in the swap or the physical memory. A process attempting to access a page currently in swap memory causes the processor to signal a page-fault interrupt. On the hardware side, the memory management unit (MMU) triggers the interrupt and translates from virtual to physical page addresses. On the software side, the *page-fault handler* reacts to page-faults by loading the requested page to the physical memory. If the physical memory is full, some other page is swapped out (cf. Fig. 2).

The verification objective is a simulation proof between a processor running a page-fault handler and user processes with virtual memory.

Implementation Model and Correctness. The correctness theorem is stated at the level of VAMP assembler combined with interleaved devices. One of the devices is instantiated with an ATA/ATAPI hard disk model. We call the processor model at the assembler level *physical machine*. It makes address translation and page-faults visible. User processes are modeled by *virtual machines* that, in essence, do not have address translation.

Theorem 9 (Virtual Memory Simulation). *The physical machine with a page-fault handler and a hard disk simulates virtual machines.*

A simulation relation projects the virtual memories of processes to the memory of the physical machine and the swap (i.e., sector) memory of the hard disk. Proving the correctness of memory virtualization is a step-to-n-steps simulation between the virtual and physical machine. The interesting case is the occurrence of a page-fault during the execution of a load or store instruction. In all other cases the semantics of the virtual and physical machines almost coincides. Thus, a proof of Theorem 9 boils down to justification of the next theorem.

Theorem 10 (Page-Fault Handler Correctness). *The simulation relation is preserved under an execution of the page-fault handler.*

The page-fault handler is a C0 program with calls to assembler subroutines, implementing the hard disk driver. The semantics of the assembler portions is encapsulated in XCalls. The extended state consists of a sector memory of the hard disk and the part of the physical memory that cannot be accessed by C0 variables. This enables us to verify the page-fault handler in our Hoare logic environment. We do not conduct all the verification even at this level. We introduce a higher-level concept for the data structures and algorithms of the page-fault handler to which we refer to as *PFH automaton*. For instance, a doubly linked list, a pointer data structure used by the page-fault handler implementation, is abstracted to a Isabelle/HOL list of references. This is formalized as an *abstraction mapping* and is proven to be preserved under page-fault handler executions.

Formal Verification. Conceptually, Theorem 10 follows from the page-fault handler functional correctness. The overall approach is sketched in Fig. 3. We first specify and prove all necessary and sufficient properties in terms of the PFH automaton. By proving in Hoare logic that the abstraction mapping holds in the state after executing the page-fault handler provided it holds before, we obtain the desired properties at the level of Simpl. Applying Theorems 3 and 4 we map these results down to the C0 big-step semantics level and further via Theorems 5 and 6 to the level of the C0 small-step semantics. We justify the XCalls to the hard disk driver by plugging in the results from [17]: driver correctness can largely be shown in a sequential setting, ignoring other devices than the hard disk. Exploiting reordering (Theorem 8) we generalize this result to arbitrary interleaved sequences. Next, by the compiling specification correctness Theorem 7 we are able to state the page-fault handler functional correctness in terms of the assembler semantics. Altogether we conclude Theorem 10.

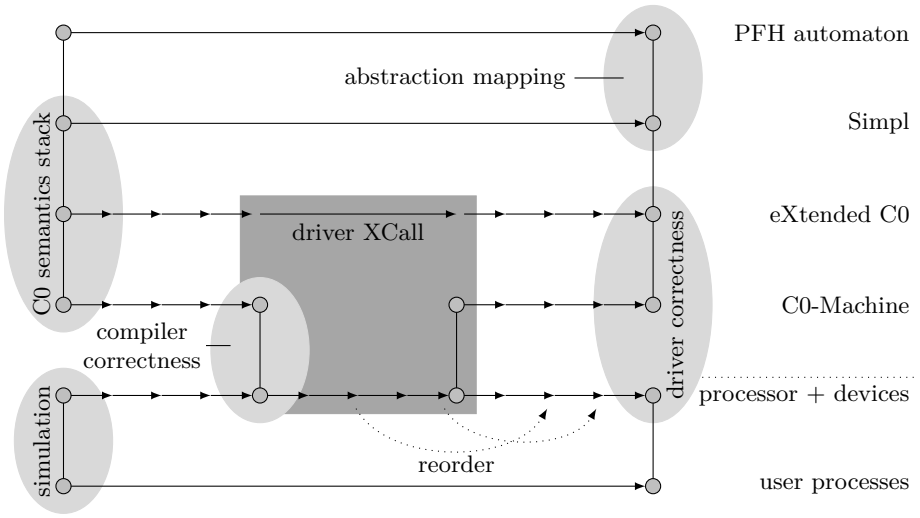

Fig. 3. Putting It All Together – Correctness of the Page-Fault Handler

8 Conclusion: Proving as an Engineering Science?

In this article we presented an overview of the core layers in the Verisoft project. They support effective reasoning at proper abstraction levels while still allowing to transfer properties down to the machine level. Our approach is quite fundamental, since every layer and all the theorems are formalized within the theorem prover Isabelle/HOL. Integrating devices into the system stack in a way that allows to preserve sequential reasoning for major parts of the system can be regarded as major achievement of the Verisoft project. The example of memory virtualization via demand paging gives strong confidence in the appropriateness of the system stack and the verification methodology.

The meta-theorems that allow us to transfer results from one level down to another only had to be proven once and for all. The effort for these proofs is compensated by the improved effectiveness in conducting major parts of the verification at a more abstract level. However, we want to point out that there is a difference between just having proven the transfer theorems (which gives an abstract notion of the soundness of the abstractions) and really instantiating them to transfer concrete properties. These instantiations can become quite tedious, since they demand to formulate the 'same' property at different abstraction levels where usually additional invariants have to be considered in order to exploit abstract information at the lower level.

All the Verisoft work done in Isabelle/HOL sums up to tens of megabytes of formal proof documents and marks the cutting edge of current academic verification projects, challenging both the theorem prover Isabelle as well as the social process to organize dozens of researchers at different places collaborating on the theories. The final correctness theorem of the page-fault handler is the

tip of an iceberg of ca. 12 MB of proof documents, importing various models, theorems, and proofs developed by different authors with different background and different style of formalization. This variety naturally results in friction losses, especially as the focus of this work was to show that one can 'get the verification done', even for a realistic system stack. The general observation is that these friction losses are annoying for the simpler proofs and tend to culminate around the more sophisticated arguments, making them even harder to obtain. For example the simulation of the C0 big-step and the intermediate small-step semantics took only about two weeks of work, since all the models were out of one hand and neatly fitted together. However, the simulation of the intermediate small-step semantics and the small-step semantics with flattened values took almost a year. This is due to some technically involved arguments and intermediate notions, especially regarding the memory update, but also due to a bunch of minor deviations in the formalization of corresponding aspects, which resulted from the fact that the theories were developed at different sites. Some were adapted and others were just bridged, because a change would have led to an enormous amount of work to accommodate the existing proofs.

An interesting social phenomenon is a kind of 'advice-resistance' and an extremely high tolerance level of the users. The lower the system level, the more invariants and assumptions appear in the theorems and proofs. These get both hard to grasp for the user and also ineffective to deal with by Isabelle's built in automation like the simplifier. To a large degree these issues can be avoided by switching from the tactical apply style to structured Isar proofs [19]. However, most users started working with apply-scripts (since the Isabelle tutorial [20] is still written in that style) and will not switch to the new paradigm until completing the proof at hand, even if that takes longer than expected.

At the ML level Isabelle provides means to profile the system. However, there is no support to effectively analyze and optimize a big theory corpus at the user level answering questions like: Where are performance bottlenecks in the proofs? Do lemmas appear in the 'right' theory or library? Are some lemmas repeatedly proven? Moreover, Isabelle's built in lemma search facility assumes that relevant theories are already loaded. This is seldom the case in large developments like ours, unless the vision of shared heaps or a proof wiki becomes reality.

One of the major challenges of Verisoft is the integration of various models, which technically means a high dependency on 'all' other theories. This lack of modularity leads to long turnaround cycles and a high sensitivity to changes of other users. Isabelle's theory structure relies on the user's discipline and does not necessarily impose proper boundaries or reflect the real dependencies. Using Isabelle's locales to disentangle theories may bring some relief, but still a more fine-grained management of the formal entities within Isabelle seems promising.

References

1. Bevier, W.R., Hunt, Jr., W.A., Moore, J S., Young, W.D.: An approach to systems verification. Journal of Automated Reasoning **5**(4) (December 1989) 411–428

2. Moore, J S.: A grand challenge proposal for formal methods: A verified stack. In Aichernig, B.K., Maibaum, T.S.E., eds.: 10th Anniversary Colloquium of UNU/IIST. Volume 2757 of LNCS., Springer (2003) 161–172
3. Neumann, P.G., Feiertag, R.J.: PSOS revisited. In: Proc. 19th ACSAC, IEEE Computer Society (2003) 208–216
4. Walker, B.J., Kemmerer, R.A., Popek, G.J.: Specification and verification of the UCLA Unix security kernel. Comm. ACM **23**(2) (1980) 118–131
5. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: Taking microkernels to the next level. SIGOPS Oper. Syst. Rev. **41**(4) (2007) 3–11
6. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Proc. 34th POPL, ACM Press (2007) 97–108
7. Tuch, H., Klein, G.: Verifying the L4 virtual memory subsystem. In: Proc. NICTA Formal Methods Workshop on Operating Systems Verification, NICTA (2004) 73–97
8. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (April 2006)
9. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: Proc. 20th TPHOLs, LNCS (2007) 189–206
10. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: The VFiasco project. In: Proc. 10th ACM SIGOPS, ACM Press (2002) 165–169
11. Tverdyshev, S., Shadrin, A.: Formal verification of gate-level computer systems. In Rozier, K.Y., ed.: LFM 2008: Sixth NASA Langley Formal Methods Workshop. NASA Scientific and Technical Information (STI), NASA (2008) 56–58
12. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. In Meltzer, B., Michie, D., eds.: Machine Intelligence 7. Edinburgh University Press (1972) 23–50
13. Daum, M., Maus, S., Schirmer, N., Seghir, M.N.: Integration of a software model checker into Isabelle. In Sutcliffe, G., Voronkov, A., eds.: Proc. 12th LPAR. Volume 3835 of LNCS., Springer (2005) 381–395
14. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In Berardi, S., Coppo, M., Damiani, F., eds.: TYPES 2003, Selected Papers. Number 3085 in LNCS, Springer (2004) 34–50
15. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In Borwein, J.M., Farmer, W.M., eds.: Proc. 5th MKM. Number 4108 in LNAI, Springer (2006) 31–43
16. Leinenbach, D., Petrova, E.: Pervasive compiler verification – From verified programs to verified systems. In: Proc. 3rd intl Workshop on Systems Software Verification (SSV). Volume 217C of ENTCS., Elsevier Science B. V. (2008) 23–40
17. Alkassar, E., Hillebrand, M.A.: Formal functional verification of device drivers. In Woodcock, J., Shankar, N., eds.: Proc. 2nd VSTTE. LNCS, Toronto, Canada, Springer (October 2008)
18. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In Ramakrishnan, C.R., Rehof, J., eds.: Proc. 14th TACAS. Volume 4963 of LNCS., Springer (2008) 109–123
19. Wenzel, M.: Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents. PhD thesis, Technische Universität München (2002)
20. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)