On the Architecture of System Verification Environments

Mark A. Hillebrand[1,⋆] and Wolfgang J. Paul[2]

[1] German Research Center for Artificial Intelligence (DFKI),
P.O. Box 15 11 50, 66041 Saarbrücken, Germany
mah@dfki.de
[2] Saarland University, P.O. Box 15 11 50, 66041 Saarbrücken, Germany
wjp@cs.uni-sb.de

Abstract. Implementations of computer systems comprise many layers and employ a variety of programming languages. Building such systems requires support of an often complex, accompanying tool chain.

The Verisoft project deals with the formal pervasive verification of computer systems. Making use of appropriate formal specification and proof tools, this task requires (i) specifying the layers and languages used in the implementation, (ii) specifying and verifying the algorithms employed by the tool chain (or, alternatively, validating their actual output), and (iii) proving simulation statements between layers, arguing about the programs residing at the different layers. Combining the simulation statements for all layers should allow to transfer correctness results for top-layer programs to their bottom-layer representation; in this manner, a verified stack can be built.

Maintaining all formal artifacts, the actual system implementation, and the (verification) tool chain is a challenging task. We call sets of tools that help addressing this task *system verification environments*. In this paper, we describe the structure, contents, and architecture of the system verification environment used in the Verisoft project.

1 Introduction

We begin with a simple question: do we know how to formally verify software? At first, the answer would be 'yes', because (i) software consists of programs, (ii) ways to formally specify program behavior can be looked up in any textbook on programming language semantics, e.g., [1, 2], (iii) it has been known since decades how to produce paper and pencil proofs for programs based on formal semantics [3, 4, 5], and (iv) these proofs could be mechanically checked by a modern computer-aided verification (CAV) system. Thus, at least in principle the problem should be solved.

However, this is an oversimplification. Software engineering does not just deal with 'programs written in a programming language' but with complex software

⋆ Work funded by the German Federal Ministry of Education and Research (BMBF) under grant 01 IS C38.

K. Yorav (Ed.): HVC 2007, LNCS 4899, pp. 153–168, 2008.

systems. These consist of many programs, which are written in different programming languages and interact with each other (and their environment) in nontrivial ways.

Thus, even the most benevolent software engineer would doubt the usefulness of software verification if programs requiring standard operating system services—e.g., file and terminal I/O, inter-process and network communication—cannot be handled. Even if such facilities could be handled, the verification results would be relative to the correctness of the underlying system and therefore questionable unless the hardware and the operating system (in particular its kernel and the device drivers) could also be verified.

In some software systems, errors have potentially disastrous consequences for body or purse and software correctness is particularly desirable. For example, security-critical systems implement cryptographic protocols to guarantee secrecy or authenticity of message exchange over untrusted networks. The systems controlling our cars, trains, or air planes are distributed and must meet hard real-time requirements.

The mission of the German Verisoft project [6] is to develop methods and an integrated set of tools permitting to handle all issues listed above and to demonstrate these by verifying entire systems of industrial interest. We call integrated sets of tools supporting the collaborative formal verification of computer systems (hardware plus software or software alone) *system verification environments*. Verification environments are themselves software systems, and like any substantial software system they should better have an architecture. This paper is about the architecture of such verification environments.

Present computer systems have a common structure: from the hardware to the applications they are organized in layers of abstraction with well-defined interfaces. For every pair of adjacent layers the lower system layer simulates the upper system layer and implements its interface. Any reasonable theory of correctness of concrete computer systems will reflect this structure. We will argue that this determines the architecture of system verification environments to a very large extent. As an example we will describe in this article the environment that was developed and is currently being used in the Verisoft project. We also announce a web site, where we expose those portions of this environment (including constructions and formal proofs) that appear sufficiently stable and do not contain confidential data of industry partners.

Overview. The remainder of this paper is structured as follows. In Sect. 2 we describe three concrete systems, which cannot be verified unless all the issues raised in the introduction are dealt with. These systems (and their requirements) were chosen together with Verisoft industry partners as concrete examples, whose complete formal verification should be made feasible by our system verification environment. Of course they will also serve as concrete examples in this paper.

Section 3 deals with computational models for describing the systems under consideration and their components. The range of these models is necessarily large, ranging from processors and devices at the low end via abstract C machines and operating systems to communicating distributed applications at the high

end. Some of these models serve as building blocks, which are referenced by concurrent and distributed models. Any system verification environment must contain formal specifications of these models. There is no complete functional correctness proof of a C program without a formal C semantics. There is no complete functional correctness proof of a driver without a formal device model. There is no complete functional correctness proof of a program making a system call unless the semantics of that call is defined somewhere.

Section 4 deals with verified components. Clearly, in a technology capable of producing verified systems it is desirable to develop a library of verified standard components *together* with their correctness proofs. Indeed, such a library is indispensable if *pervasive* system verification [7, 8] is attempted, i.e., the verification of entire systems across several layers of abstraction. It turns out that that standard components often provide a simulation in terms of the models from Sect. 3. Processors for instance simulate assembler programs by registers, memories, and gates. Compilers translate source programs into target programs simulating the source programs.

In Sect. 5 we consider another hierarchy different from the hierarchy of system layers, namely the hierarchy of semantic models. In its basic form, this hierarchy and its associated soundness results are classical material from textbooks on programming language semantics [1]. We consider small step semantics, big step semantics, and Hoare logic. We use small step semantics in our system models where we need to argue about communicating systems, sometimes doing rely / guarantee style proofs [9]. Big step semantics and Hoare logic are equivalent and allow to prove pairs of pre and post conditions. Because the abstraction level is higher than in small step semantics, proofs can be generated with higher productivity than in small step semantics; for the Hoare logic we also make use of a verification condition generator.

In addition to program state and functions, we also allow abstract state and functions at the Hoare logic layer. This way, proofs in the Hoare logic may be conducted relative to low-level functions or libraries. Consider a driver writing some C variables to a disk. Although that driver has in line assembler code (otherwise it cannot access the ports of the hard disk controller) its effect can be specified in the Hoare logics by a pre and post condition pair operating on abstract state representing the disk configuration. Hoare logics of this kind we call *extended Hoare logics*; soundness results for such Hoare logics are relative to the postulated extended semantics.

Section 6 deals with proof tools. There must be a combination of interactive and automatic proof tools. Automatic tools increase productivity, thus they cannot be ignored in an engineering effort. Because the complete verification of entire systems is out of reach for present automatic tools, at least one 'general purpose' interactive prover must be present. We mainly use Isabelle/HOL [10] as general purpose prover. Isabelle/HOL also serves as an integration platform for most automatic proof tools.

Section 7 we shortly describe how the contents of the system verification environment are stored and related to each other in a version control system.

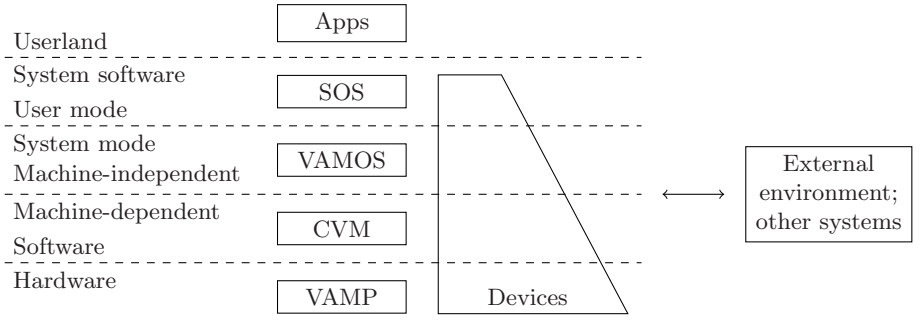

Fig. 1. Implementation layers of the academic system (Verisoft subproject 2)

Also, we describe the portions of the environment which are currently made public. We summarize in Sect. 8.

2 Overview of Systems

In this section, we describe three systems for which the formal pervasive verification is attempted in the Verisoft project: the 'academic system' (Verisoft subproject 2), the 'chipcard-based biometric identification (CBI) system' (Verisoft subproject 4), and the 'automotive system' (Verisoft subproject 6). All these systems use the same implementation languages and also share significant parts of the hardware and system software implementation. In particular, the employed hardware architecture and the architecture-specific parts of the microkernel implementation are reused for all described systems.

Academic system. The academic system is a computer system for writing, signing, and sending emails. It covers all implementation layers from the gate-level hardware to communicating concurrent programs and thus represents a vertical cross section of a general-purpose computer system.

Let us describe the components of the academic system in bottom-up fashion (see also Fig. 1). The lowest layer of the academic system consists of a *hardware architecture*, featuring the VAMP, a DLX-like processor with address translation, and abstractions of memory-mapped I/O devices (timer, network interface card, keyboard, terminal, and hard disk). The next layer of *communicating virtual machines (CVM)* establishes a hardware-independent programming interface for a microkernel and a virtual computation environment for concurrently running processes. Some parts of CVM must be implemented in assembler since C0, a subset of regular C, lacks low-level programming constructs. The *microkernel*, which is called VAMOS, is based on the CVM interface and contains no assembler parts. On the next-higher layer the *simple operating system* (SOS) is located, which runs as a (privileged) user process. It offers file I/O, inter-process communication, sockets, and remote procedure calls to user processes. Last but

not least several user processes are needed to implement the desired functionality of the academic system; these include a signing software, an SMTP client (and, on the receiving side, an SMTP server), and a simple mail user agent.

Chipcard-based biometric identification system. The chipcard-based biometric identification (CBI) system is an access solution, which grants system access based on the similarity between fresh biometric data and reference biometric data. Fresh biometric data is sampled using a biometric sensor. The reference biometric data is read from a chipcard, which belongs to the user. Biometric data is considered personal data in Germany and has to be kept confidential in accordance with German privacy regulations. Additionally, communication between the host system and the user's chipcard must have certain cryptographic properties like authenticity and integrity of messages. This is established by running a cryptographic protocol between host and chipcard.

Automotive system. The automotive system is the prototype of an automatic emergency call system, which is meant to contact the public-safety answering point (PSAP) automatically in case of a (severe) car crash. The system is realized as a distributed system, namely a cluster of electronic control units (ECUs) connected to each other via a shared serial bus. Bus communication is time-triggered, i.e., access to this bus is granted to the individual ECUs according to a static, periodical schedule. The schedule period is called a *round*; rounds are evenly divided into *slots*, which represent the minimal bus allocation intervals. In each slot, the sending ECU may broadcast a *frame* to all other ECUs. Frames contain *messages* as a payload. Messages have types. On each ECU runs a small operating system, which is called OLOS. OLOS maintains a buffer for each message type. Incoming messages are stored into this buffer and outgoing messages are transmitted from that buffer (according to another, static schedule). Applications may access the message buffer using system calls. The user view of the whole cluster is as follows: applications are executed on all ECUs in lock-step while they seem to communicate over shared variables.

In order for this hardware / software stack to work as specified, the following two aspects concerned with timing are crucial. Already at the hardware level, a clock synchronization algorithm must be used to compensate for the different hardware clock frequencies of the ECUs (due to manufacturing tolerances or environmental conditions). Otherwise, ECUs will violate slot and round boundaries, in the long run causing bus contention and compromise of the communication mechanism. At the software level, for the system to operate in lock-step fashion, both the system software and the applications must run fast enough to observe slot and round boundaries. To show that these constraints are met, a worst-case execution time analysis for all the software is necessary.

3 Computational Models

The repository of computational models plays a very central role in the system verification environment. Typically, each model is referenced in three situations:

(i) in correctness proofs for programs in this model, (ii) in simulation theorems showing that the model is simulated by a model from an adjacent *lower* system layer, and (iii) in simulation theorems showing that the model simulates a model belonging to an adjacent *higher* system layer (which may also be a program correctness proof). In the remainder of this sections, we shortly sketch standard models (for languages and systems without devices), devices models (for systems with devices), and distributed models (for systems communicating over devices).

Standard models. We consider the following standard models for languages: (i) the model of the machine language / instruction set architecture (ISA) of the VAMP [11, 12], a variant of the DLX architecture [13], (ii) the model of VAMP assembler, (iii) the semantics of C0, which is the subset of C we use [14], and (iv) the semantics of C0A, which is C0 with in line assembler code. In the latter model, we have to consider both the computation of a (compiled) C0 program and of an assembler program. These computations influence each other in cases where in line assembler instructions update C0 variables, e.g., when a processor register is copied to a C0 variable. This requires knowledge about the memory layout employed by the C0 compiler, which allocates C0 variables to VAMP memory ranges.

Standard system models are obtained by combining one (or more) of the above models with specifications of special operations (e.g., system calls). In the academic system, we consider the following system models: (i) Communicating virtual machines (CVM), a generic model of operating system kernels permitting to abstract from the use of in line assembler in the lower-level kernel implementation [15]. This is a concurrent model of computation consisting of an abstract kernel modeled as a C0 program and user processes modeled as VAMP assembler programs. At any time either the kernel or one of the user processes are running. The kernel is non preemptive and may only be interrupted by reset. If interrupts occur during user process execution, the kernel is entered. No memory is shared between user processes or (C0 variables of) the abstract kernel. (ii) VAMOS, the abstraction of an instantiation of CVM with a specific kernel [16]. User processes may be assembler programs, or, in an extension of this model, also C0 programs. In the latter case, we also abstract from the concrete VAMOS scheduler. (iii) The model of the simple operating system (SOS), which specifies the system calls provided for C0 or assembler user applications.

Device models. With the standard models above we cannot yet handle the numerous situations where I/O is performed in the academic system (swap memory, terminal, file, and network access). This makes a generalization of the hierarchy above necessary. We have to define models for specific devices for use in the specific layers of the model stack. Depending on the layer and its level of abstraction, even variants of models for a device may be required. From one layer to the next, a more abstract variant of a device model is employed when the lower layer implements a (nontrivial) driver for that device.

We distinguish the following models. (i) Hardware with devices. Devices employed at this stage may be gate-level implementations of devices and, if this

is the case, must be part of the hardware correctness proofs. (ii) Instruction set architecture with devices. For special cases (e.g., hard disk) this may be a nonconcurrent model [17]. In the general case, this model is nondeterministic, concurrent, and, when communication between computer systems is considered, also distributed, cf. [18, 19]. The obvious next two models are (iii) assembler with devices permitting to program device drivers in assembler (e.g., [20]) and (iv) C0A with devices permitting to use these drivers in C0A programs. (v) As a next step we turn the drivers into external functions. This permits to return to the syntax of ordinary C0. For nontrivial drivers (which are not only used to expose devices directly to user-level device drivers), more abstract variants of a device may be employed. For example, we abstract a hard disk used for swapping into a 'swap memory', which by the page fault handler via driver calls to swap in or swap out pages. This model we call C0 with devices.

In the system models devices can only be accessed by (or through) the kernel; device ports are never mapped to user memory and interrupts are relayed over the kernel. For each system model, we have an extended model with devices. From one system model to the next, certain devices may be hidden completely, e.g., the timer used by the microkernel's scheduler is not visible to the upper layers.

Extensions for distributed and communicating systems. In the automotive system several processors together with their bus interfaces form a cluster of electronic control units (ECUs), which communicate over a FlexRay like shared serial bus. A technically interesting complication arises from the fact that each ECU has a private oscillator with a clock period close but not equal to a reference clock period. As a consequence the bus interfaces contain serial interfaces and hardware implementing a clock synchronization algorithm [18, 19].

Thus, in the automotive system, the following distributed models are considered: (i) A distributed hardware model, which extends the usual digital hardware model in two ways. The hardware of the entire system is partitioned into portions with the same oscillator (the ECUs). Moreover, for the drivers and registers directly connected to the bus set up and hold times (and metastability of flipflops) must be considered [19,21]. (ii) A distributed ISA model with FlexRay like devices modeling the communicating ECUs at the ISA level. (iii) A distributed assembler model with FlexRay like devices modeling the communicating ECUs at the assembler level. (iv) Employing the real-time kernel OLOS (OSEKtime like operating system) [22], which provides access to the FlexRay to C0 user applications, we obtain a distributed OLOS model. (v) The top level model used in the automotive system is a model of communicating automata called AutoFocus Task Model (AFTM) [23].

The model for academic systems communicating over the Internet is slightly simpler because only discrete systems are considered. From any computational model with network interface card as device, a distributed version of the model can be derived. This is simply the distributed system consisting of copies of the basic model and the model for their connection, i.e., the Internet, which includes a model of packet loss.

4 Verified Components

Based on models of computation of the previous section we may now proceed to present a library of verified components. A verified component has five parts: (i) a formal specification of the component, which defines user visible data structures and operations, (ii) a formal specification of the model in which the component is implemented, (iii) formal specifications of subcomponents used in the construction of the component (if any), (iv) an implementation of the component, and (v) a formal proof that the construction meets the specification. Formal specifications used in components may coincide with the specification of the computational models from Sect. 3, e.g., for components implementing system layers. Currently, a large number of verified components is under development or completed. For the components listed below at least the first four parts are completed and the formal proof is either completed or under construction.

Hardware. (i) VAMP processors built from ordinary hardware (i.e., gates, registers, and memories). Formal proofs against the VAMP ISA are completed in PVS [12,24] and under construction in Isabelle/HOL. (ii) processors with devices constructed from ordinary hardware devices specified at the hardware level. The specification of such processors is given by the computational model of VAMP ISA with devices. The correctness proof is a not completely obvious extension of 'ordinary' processor correctness theorems, because the ISA model of a processor with one or more devices is in general distributed and nondeterministic; the nondeterminism is resolved by the implementation's timing behavior. (iii) For the automotive system, interfaces for a FlexRay like bus have been constructed at the gate-level. The correctness proof for these devices is conducted in the distributed hardware model described above. Both the correctness of a serial interface and the implementation of a clock synchronization algorithm in hardware have to be shown. For paper and pencil proofs see [18, 19]. The part of the formal correctness proof dealing with setup and hold times of registers is reported in [21].

Basic data structures and algorithms. For use by other C0 programs, we currently provide three libraries of basic data structures and algorithms: a library for doubly linked lists, a string library [25], and a big number library [26]. All libraries are programmed in C0. Specification and correctness proofs of the library functions are done in the Hoare logic for C0. The list library is used by the other two libraries.

Compiler. (i) The C0 compiler (backend) translates abstract syntax trees of C0 programs into VAMP assembler programs [14]. It is specified in C0 small steps semantics and uses the list library. The correctness is shown using C0 Hoare logic. Correctness with respect to small steps semantics is inferred using the soundness of the Hoare logic (cf. Sect. 5). (ii) A fairly straightforward extension, assuming 'acceptable' behavior of in line assembler portions, gives the correctness of the C0A compiler. (iii) A more involved extension of the compiler is a copying garbage collector [27], which is crucial for certain (application) code,

such as the big number library. (iv) The simulation of the model C0A with devices by the model assembler with devices works only under certain software conditions: compiled C0 instructions, e.g., of the kernel, must not be interrupted and devices may only be accessed with in line assembler and not by compiled C0 statements. This guarantees that the execution of compiled C0 statements and device transitions do not interfere with each other. Note that user programs never directly access devices in our systems and can be interrupted, which has the effect of the non-interruptible kernel taking over control.

Device drivers. (i) Elementary device drivers are pieces of assembler program copying data between a region of memory in the processor and a device specific region of memory on the device, e.g., disk space. Elementary device drivers are specified and programmed in the model assembler with devices. For a paper and pencil correctness proofs of elementary device drivers for a disk and a UART see [17, 20]. (ii) Elementary device drivers may be embedded into functions of a C0A program. As these device drivers usually abstract from their assembler implementation, their specification can be done relative to the C0 model with devices. Typically, these drivers provide an abstracted view of the device they control. Note that for interfacing reasons, the correctness proofs of any C0A programs has to refer C0 calling convention and memory layout. (iii) User-level devices are implemented using system calls for device access provided, e.g., by the model VAMOS with devices. They may be verified relative to the specifications of these system calls in an extended Hoare logic (cf. Sect. 5). An example of such a driver is the hard disk driver used by the simple operating system to implement a simple file system. In contrast to the elementary device driver used for swapping, this disk driver is interrupt-driven.

System software. The specification of the generic operating system kernel CVM is directly given by the model CVM with devices. The so called concrete kernel, which is the CVM implementation for a given abstract kernel, is obtained by linking and compiling the abstract kernel with a C0A program. This program implements the CVM functionality. Its major data structures are the process control blocks and the page tables. Its major functions are swap memory management, page fault handling, context switching, and operations on the user assembler machines, such as user memory copy operations [15]. As the kernel is non-preemptive parts of its correctness (in particular the page fault handler's) can be shown in an extended Hoare logic (cf. Sect. 5).

VAMOS [16] is an instantiation of CVM, which was inspired by the L4 microkernel [28]. It calls CVM functions and is therefore implemented in the model CVM with devices. Proofs can make use of extended Hoare logics relative to a specification of this model. Thus, e.g., the correctness of inter-process communication (IPC) operations, which implement a rendezvous protocol, may be shown relative to the correctness of the user memory copy operations provided by CVM. In the VAMOS model allowing C0 user programs, two additional abstractions have to be justified. First, the scheduler is abstracted away. This requires to prove fairness of the scheduler. Second, some user processes are allowed to be C0 programs, which must be linked against a system call library implemented

in C0A. Proving the library correct, requires another application of compiler correctness in the correctness proof.

The implementation of the simple operating system makes use of VAMOS, drivers for disk, terminal, and network interface and an implementation of TCP. On top of the disk driver, a simple file system is implemented. On top of the TCP implementation, socket management functions have been implemented.

To implement remote procedure calls (RPC) for SOS applications a port mapper, user-level primitives for the implementation of remote procedure call (RPC) on top of SOS, and an interface compiler have to be provided [29]. SOS and the applications running under it form a distributed system. Computations of user processes and of SOS are interleaved. Correctness proofs about applications interacting via RPC thus need to use rely / guarantee arguments (e.g., [30]).

Applications. Applications in the academic system are (i) an SMTP client and server for email transfer [30], (ii) a signature server used to sign electronic mails and (iii) an email client, which uses the previous applications and implements a user interface [31]. The applications run under SOS, make use of SOS RPC and the SOS file system. The signature server also makes use of formally verified cryptographic primitives (e.g., [32]). For the biometric identification system, security properties of the cryptographic protocol have been formally modeled and proven using VSE [33, 34]. Certain properties of the emergency call application in the automotive system were formally verified in the AutoFocus task model (AFTM) using the AutoFocus tool [23].

5 Semantics Hierarchy

Because we need to consider interleaved programs in several places of the Verisoft project (e.g., RPC clients and servers), the standard models listed in Sect. 3 are *all* small steps semantics. Although in the end we need many program correctness theorems with respect to small steps semantics, we produce the correctness proofs as much as possible using the verification condition generator (VCG) of a Hoare logic for C0 [35]. This needs to be justified by a hierarchy of C0 semantics, which is also part of the system verification environment: the small steps semantics of C0 (Sect. 3), a big steps semantics for C0 [35], and a Hoare logic with the VCG mentioned above as a proof tool.

In order to go back and forth between the three levels of C0 semantics we have proven formal versions of classical textbook theorems. First, the soundness of the big step semantics with respect to the small steps semantics. Second, the equivalence between Hoare logic and big step semantics. Because of the shallow embedding of the Hoare logic into Isabelle/HOL there can be no general (program-independent) equivalence proof in Isabelle because such a proof would require to quantify over all types of Isabelle [35, Chapter 8]. We expect, however, that the proofs obligations for individual C0 programs can be automatically proven.

We also use an extended version of this semantics stack with which (noninterrupted) C0 programs may be verified in the Hoare logic relative to an abstract specification, e.g., a system model providing system calls. This may be used in

the verification of drivers (user or kernel level), abstract kernels, or user applications. The abstract specification is represented in the Hoare logic with ghost (i.e., non-program) variables and postulated pre and post condition pairs. All assumptions made in this manner need of course to be verified; the soundness results for this semantics are only relative to the assumed specifications. These assumptions must be discharged when transferring concrete properties from the Hoare logic to the small steps semantics.

6 Proof Tools

In our system verification environment we use interactive provers (mainly Isabelle/HOL [10]) as the central platform for formal modeling and verification tasks. All computational and semantic models are expressed as Isabelle/HOL theories. Components (hardware and software) to be verified are usually written relative to one of the semantic models in a deep embedding. There are two important exceptions to this rule. In the C0 Hoare logic, C0 expressions are shallow embedded to improve verification productivity [35]; special care must be taken about soundness here, as mentioned earlier. Also, hardware models are formulated in a synthesizable subset of Isabelle/HOL [36]. Since gate-level hardware constitutes the bottom of our model stack, we cannot show soundness here.

The benefit of having a general purpose interactive theorem prover as a central component is that there is always a verification tool of last resort when automatic verification fails. However, increasing automation is clearly the key to success in the verification of industrial computer systems. We have integrated a number of automatic proof tools into Isabelle/HOL via Isabelle's oracle interface. These tools either hook into one of the semantic model described above or Isabelle/HOL directly. We trust the tools to produce correct results; hence, currently, no proof objects are imported into Isabelle for automatically proven goals.

Proof tools that have been integrated into Isabelle include classical symbolic model checkers [36], software model checkers and shape analysis tools [37, 38, 39, 40], translation validation tools [41], and first-order logic theorem provers [42,43]. As mentioned earlier, the C0 Hoare logic includes a verification condition generator [35]. For the automotive system, we use AbsInt's worst-case execution time analyzer aiT [44] based on abstract interpretation, which is, however, not directly integrated into Isabelle/HOL.

7 Repository Implementation and Public Releases

To form a viable platform for the development of system correctness proofs, we keep all afore-mentioned artifacts (computation models, proof objects, tools, etc.) in a central repository. We make use of the version control system Subversion [45], which provides revision tracking and concurrent operation in an easy-to-use fashion. In addition to the artifacts relevant for the formal verification, we also store system implementations, the development tool chain, and additional documentation in the repository. All of these items are organized

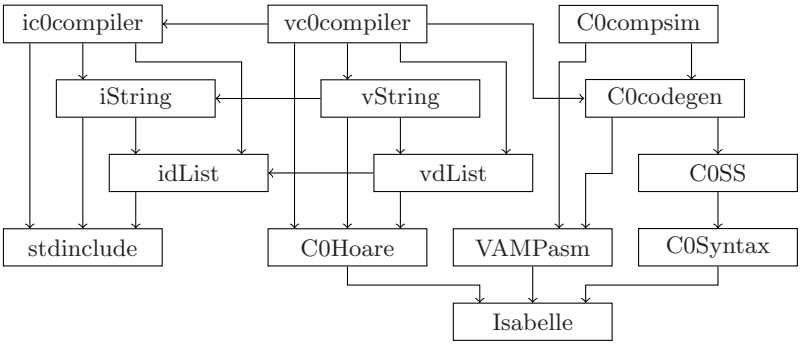

Fig. 2. Example of modules and their dependencies

in so-called *modules*, often on a more fine-grained level than described earlier (e.g., to share commonly used definitions or results across modules). Modules are related to each other via *dependencies*, which have to be acyclic.

Figure 2 shows a number of modules from our repository, which are related to the verification of our non-optimizing C0 compiler [14]. The four boxes on the left-hand side represent the implementation modules of the compiler: the compiler itself, the libraries that it needs, and standard headers. The latter need not to be verified. For the other three implementation modules, there is a corresponding code verification module. All proofs therein are conducted in the C0 Hoare logic [35], which is implemented in Isabelle/HOL [10]. In the top-level code verification module, vc0compiler, the output of the C0 compiler implementation is shown to be equivalent to the output of an (abstract) code generation algorithm [46]. This algorithm maps syntax trees of C0 programs to VAMP assembler programs, whose specifications are both modeled in Isabelle/HOL. In the module C0compsim, the correctness of the code generation, expressed as a simulation theorem over C0 and VAMP assembler computations, is shown.

We will make available self-contained portions of the repository, which appear to be appear sufficiently stable and do not contain confidential data of industry partners. Currently, four releases have been made public.[1] Two of the releases deal with the code-level verification of the C0 string library [25] and the C0 compiler [46], covering all the modules shown in Fig. 2 except C0compsim, which is planned to be released next. As mentioned above, the code verification is conducted in the C0 Hoare logic verification environment. For this purpose, the C0 implementations in concrete syntax have been translated into their Hoare logic representation. The translator is also included in the latest release. In the C0 Hoare logic, Hoare triples for partial and total correctness have been shown. In addition to the functional correctness and termination, the absence of certain runtime errors has been proven (e.g., integer overflows and out-of-bounds array access). These properties would be required at a later stage to translate total

[1] `http://www.verisoft.de/VerisoftRepository.html`

correctness results at the Hoare logic level down to our lower-level semantics, i.e., in the end to compiled program running on the target architecture.

The third release deals with the code-level verification of the C0 big integer library, implementing arbitrary-precision integer operations based on a linked-lists representation of integers. Supported operations include addition, subtraction, multiplication, division, remainder, and exponentiation modulo an integer.

The fourth release deals with the code-level verification of the email client of the academic system relative to the services provided the operating system and applications for signing and email transfer [31]. In addition to modules described earlier, it contains modules for the email client implementation and proofs.

8 Summary

We have presented an overview of the system verification environment used in the Verisoft project to carry out the formal pervasive verification of entire systems of industrial interest. The architecture of our verification environment is to a large extent determined by each system's architecture and its requirements. The system's layers, its implementation languages, its components, and its tool chain are all represented in the verification environment, thus enabling to formally reason on system requirements. The form of the representations is on the one hand shaped by the system requirements and on the other hand by verification productivity concerns: we are employing small-steps semantics to reason on concurrent, communicating programs, but we switch to more abstract semantics (for which we have verification condition generation and integration of automatic provers) wherever possible. Soundness and simulation theorems of the higher-level relative to lower-level semantics justify this approach. Thus, in addition to the stack of computational models, which inherit from the system implementation structure, a semantic stack is build. We have announced a web site, where we have started to publish portions of our verification environment.

References

1. Winskel, G.: The formal semantics of programming languages. MIT Press, Cambridge (1993)
2. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. John Wiley & Sons, Chichester (1992)
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM 12, 576–580 (1969)
4. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Comm. ACM 18, 453–457 (1975)
5. Gries, D.: The Science of Programming. Springer, Heidelberg (1987)
6. The Verisoft Project. `http://www.verisoft.de/`
7. Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. Journal of Automated Reasoning 5, 411–428 (1989)

8. Moore, J S.: A grand challenge proposal for formal methods: A verified stack. In: Aichernig, B.K., Maibaum, T.S.E. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 161–172. Springer, Heidelberg (2003)
9. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Univ. Press, Cambridge, UK (2001)
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
11. Mueller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer, Heidelberg (2000)
12. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, Springer, Heidelberg (2003)
13. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco (1996)
14. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: Aichernig, B., Beckert, B. (eds.) SEFM 2005, pp. 2–11 (2005)
15. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: Hurd, J., Melham, T.F. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
16. Dörrenbächer, J.: (VAMOS microkernel, formal models and verification) Talk given at the International Workshop on System Verification, SV 2006, Sydney, Australia (August 7–8, 2006), `http://www.cse.unsw.edu.au/~formalmethods/events/svws-06/VAMOS_Microkernel.pdf`
17. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD 2005, pp. 309–316. IEEE Computer Society, Los Alamitos (2005)
18. Knapp, S., Paul, W.: Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 53–81. Springer, Heidelberg (2007)
19. Knapp, S., Paul, W.: Pervasive verification of distributed real-time systems. In: Broy, M., Grünbauer, J., Hoare, T. (eds.) Software System Reliability and Security, Trento, Italy. NATO Security Through Science Series. Sub-Series D: Information and Communication Security, vol. 9, pp. 239–297. IOS Press, Amsterdam, Trento, Italy (2007)
20. Alkassar, E., Hillebrand, M., Knapp, S., Rusev, R., Tverdyshev, S.: Formal device and programming model for a serial interface. In: Beckert, B. (ed.) Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany, pp. 4–20 (2007)
21. Schmaltz, J.: A formal model of lower system layer. In: Gupta, A., Manolios, P. (eds.) FMCAD 2006, pp. 191–192. IEEE Computer Society, Los Alamitos (2006)
22. Knapp, S.: Towards the verification of functional and timely behavior of an eCall implementation. Master's thesis, Saarland Univ. (2005), http://www-wjp.cs.uni-sb.de/publikationen/Knapp05.pdf
23. Botaschanjan, J., Gruler, A., Harhurin, A., Kof, L., Spichkova, M., Trachtenherz, D.: Towards modularized verification of distributed time-triggered systems. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 163–178. Springer, Heidelberg (2006)

24. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 301–316. Springer, Heidelberg (2005)
25. Starostin, A.: Formal verification of a C-library for strings. Master's thesis, Saarland Univ. (2006), http://www-wjp.cs.uni-sb.de/publikationen/St06.pdf
26. Fischer, S.: Formal verification of a big integer library including division. Master's thesis, Saarland University (2007)
27. Appel, A.W., Ginsburg, M.: Modern Compiler Implementation in C. Cambridge Univ. Press, New York (1998)
28. Liedtke, J.: On micro-kernel construction. In: SOSP 1995. Proceedings of the 15th ACM Symposium on Operating systems principles, pp. 237–250. ACM Press, New York (1995)
29. Shadrin, A.: Design and implementation of the portmapper and RPC primitives in the context of the SOS. Master's thesis, Saarland Univ. (2006), http://www-wjp.cs.uni-sb.de/publikationen/Sh06.pdf
30. Langenstein, B., Nonnengart, A., Rock, G., Stephan, W.: A history-based verification of distributed applications. In: Beckert, B. (ed.) Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany, pp. 70–84 (2007)
31. Beuster, G., Henrich, N., Wagner, M.: Real world verification – Experiences from the Verisoft email client. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) ESCoR 2006. CEUR Workshop Proceedings. CEUR-WS.org, vol. 192, pp. 112–125 (2006)
32. Lindenberg, C., Wirt, K., Buchmann, J.: (Formal proof for the correctness of RSA-PSS) Cryptology ePrint Archive, Report 2006/011, http://eprint.iacr.org/2006/011
33. Cheikhrouhou, L., Rock, G., Stephan, W., Schwan, M., Lassmann, G.: Verifying a chipcard-based biometric identification protocol in VSE. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 42–56. Springer, Heidelberg (2006)
34. Hutter, D., Langenstein, B., Sengler, C., Siekmann, J.H., Stephan, W., Wolpers, A.: Verification Support Environment. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 476–493. Springer, Heidelberg (2005)
35. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technical University of Munich (2006)
36. Tverdyshev, S.: Combination of Isabelle/HOL with automatic tools. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 302–309. Springer, Heidelberg (2005)
37. ACSAR: Automatic Checker of Safety properties based on Abstraction Refinement. http://www.mpi-inf.mpg.de/~seghir/ACSAR/ACSAR-web-page.html
38. ARMC: Abstraction refinement-based model checker for safety and liveness properties. http://www.mpi-inf.mpg.de/~rybal/armc/
39. Jahob and Bohne: Verifying data structure consistency. http://www.mit.edu/∼vkuncak/projects/jahob/
40. Daum, M., Maus, S., Schirmer, N., Seghir, M.N.: Integration of a software model checker into Isabelle. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 381–395. Springer, Heidelberg (2005)
41. Emeliyanenko, P.: Automatic verification of conditions for absence of interrupts. Bachelor's thesis, Saarland Univ. (2006) http://react.cs.uni-sb.de/fileadmin/user_upload/react/theses/PEmeliyanenko.pdf
42. SPASS: An Automated Theorem Prover for First-Order Logic with Equality. http://spass.mpi-sb.mpg.de/

43. e-SETHEO prover system. http://www4.in.tum.de/~stenzg/
44. Ferdinand, C., Heckmann, R.: Verifying timing behavior by abstract interpretation of executable code. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 336–339. Springer, Heidelberg (2005)
45. Subversion: An open-source revision control system. http://subversion.tigris.org/
46. Petrova, E.: Verification of the C0 Compiler Implementation on the Source Code Level. PhD thesis, Saarland University, Computer Science Department (2007)