Model Stack for the Pervasive Verification of a Microkernel-based Operating System*

Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan

Saarland University, Computer Science Dept.
66123 Saarbrücken, Germany
{md11,jandb,sebastian}@wjpserver.cs.uni-sb.de

Abstract. Operating-system verification gains increasing research interest. The complexity of such systems is, however, challenging and many endeavors are limited in some respect: Some projects focus on a particular aspect like memory safety, not pursuing functional correctness. Others restrict their verification efforts to a single layer of software, assuming correctness of those below. Only few projects aim at pervasive formal verification of a computer system over several software layers.
In our paper, we present an approach to the formal specification of a microkernel-based operating system at several layers and glance on our verification experience with this model stack. From our experience, we conclude that pervasiveness entails more than just cumulative verification efforts on several layers. In fact, it is a challenging task to integrate models and proofs into a uniform, coherent theory.

1 Introduction

Software-verification tools have greatly improved in recent years. As a consequence, the verification of low-level software gains increasing research interest as a touch-stone for the industrial applicability of software verification in general. While many aspects of operating-system verification are currently studied, they are not integrated into a uniform, coherent theory. In our paper, we concentrate on the integration of several operating-system models in order to form a pervasive verification stack. We point out the integration problems and describe the specific design decisions that we made in order to support integration.

A characteristic problem of operating-system software is that hardware components become visible and the program functionality cannot be expressed in the pure semantics of a high-level language like C. Usually, this functionality is implemented in assembly and encapsulated in a small number of C functions, or *primitives*. Thus, only a small part of low-level code is indeed implemented in assembly while larger code portions are written in C and just use these primitives. From a model-theoretic point of view, such primitives extend the original C semantics. Another important problem related to operating systems regards the atomicity of operations in a concurrent setting. The processor might be interrupted in its current computation after each *assembly instruction* but a single C statement is usually compiled into several instructions.

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Consequently, we need an assembly semantics, a C semantics and a simulation theorem between both semantics. Leinenbach and Petrova [1] provide exactly that: a proven correct compiler that translates programs from the perfectly type-safe C variant *C0* into the assembly language of our RISC processor. An important feature of C0 is the inline-assembly statement, which permits the language extensions via primitives. Using this mechanism, we have implemented our operating system in C0. The correct C0 compiler is an elementary prerequisite for our model stack.

Outline. In Sect. 2, we explain the fundamentals of our operating system. This section comprises a sketch of the correct compiler, a survey of the features of our microkernel, and an outline of the system's implementation design. With these prerequisites in place, we give an overview of our model stack in Sect. 3. Certainly, we cannot descend into all details of the stack in this paper.[1] Hence, we confine ourselves to aspects of encapsulation and state partitioning. In Sect. 4, we describe our formal specification of the generic interface between the user processes and the kernel. Moreover, we sketch our formal model for assembly and C0 processes. This process abstraction is an elementary prerequisite for the model of *communicating user processes* as well as for the top-level specification of the operating system. We illustrate in Sect. 5, why we use the process abstraction already in the VAMOS specification and how we have constructed the latter model in order to support the abstraction towards COUP and the top-level specification. We conclude in Sect. 6.

2 Background

2.1 A Correct Compiler for an Extensible Language

In this section, we summarize work of Leinenbach and Petrova [1,2]. They provide a formally verified compiler that translates programs from the perfectly type-safe C variant C0 into the assembly language of a RISC processor. Restrictions of C0 in comparison to C are that expressions must be side-effect-free, all type conversions have to be made explicitly, and there is no pointer arithmetic. In spite of that, C0 still features dynamic memory allocation and inlined assembly.

The inline-assembly statement is an important feature for language extensions. Though the effect of this statement cannot be expressed in the formal C0 semantics, the compiler literally embeds the given assembly code into the compilation. Using this hook, we can implement C0 functions that use inline assembly in order to extend C0 by a functionality that cannot be expressed by other C0 statements.

[1] The complete Isabelle/HOL theories of the models as well as detailed documentation are made available at the Verisoft repository, see <http://www.verisoft.de/VerisoftRepository.html>

For C0 and for the assembly language, two small-step transition semantics have been formalized in Isabelle/HOL. Leinenbach has formally proven compiler correctness by a stepwise simulation between both semantics. Below, we describe the semantics in detail. We often deal with structured values, which we define by enumerating the components in prose, e. g., "a value x consisting of two components *this* and *that*". We refer to a single component with a dot, e. g., $x.\mathit{this}$ refers to component *this* of value x. An update of this component is denoted by $x\,[\mathit{this} := q]$.

C0 Semantics. For lack of space, we can only glance at the C0 semantics. C0 programs are statically represented by the program environment Γ, which comprises a symbol table of global variables, a type-name environment, and a function table. The dynamically changing state s_{C0} of a C0 program in execution is composed of

- the remaining program $s_{\mathrm{C0}}.\mathit{prog}$, and
- the current state of the program variables $s_{\mathrm{C0}}.\mathit{mem}$.

In the following sections, we assume an evaluation function *get-val* for the look-up, and an update function *set-val* for the manipulation of a certain variable in the memory. We refer to the value of expression e in state s_{C0} by $\mathit{get\text{-}val}(s_{\mathrm{C0}}, e)$. If we update the left-value l in state s_{C0} with some expression u, we denote the resulting configuration by $\mathit{set\text{-}val}(s_{\mathrm{C0}}, l, u)$.

The transition relation $\delta^{\mathrm{seq}}_{\mathrm{C0}}$ of this semantics is a partial function.

The Target Assembly Language. The assembly semantics was developed for the RISC processor VAMP [3]. The assembly semantics abstracts from the paging mechanism of the processor and employs a linear memory model. An assembly state s_{asm} consists of the following components:

- the normal and the delayed program counters, $s_{\mathrm{asm}}.\mathit{pc}$ and $s_{\mathrm{asm}}.\mathit{dpc}$, respectively, implementing the delayed branch mechanism.
- the general-purpose register file $s_{\mathrm{asm}}.\mathit{gpr} \in \{0,\ldots,31\} \to \{0,1\}^{32}$.
- the memory size $s_{\mathrm{asm}}.V$ measured in pages of 4096 bytes. It defines the set of available memory addresses: $\mathit{VA}(s_{\mathrm{asm}}) = \{a \mid a < s_{\mathrm{asm}}.V \cdot 4096\}$
- the byte-addressable linear memory $s_{\mathrm{asm}}.\mathit{vm} \in \mathit{VA}(s_{\mathrm{asm}}) \to \{0,1\}^8$

We denote the state space of the assembly semantics by $\mathcal{S}_{\mathrm{asm}}$. Assembly computations are modeled by the partial function $\delta^{\mathrm{seq}}_{\mathrm{asm}} \in \mathcal{S}_{\mathrm{asm}} \rightharpoonup \mathcal{S}_{\mathrm{asm}}$. Note that the effects of exceptions like illegal page faults cannot be fully determined from the assembly-machine state. In that case, $\delta^{\mathrm{seq}}_{\mathrm{asm}}$ gets stuck. But with sufficient resources, a compiled C0 program does not generate exceptions during normal execution. Moreover, the memory size $s_{\mathrm{asm}}.V$ can neither be read nor changed by the assembly machine itself but depends on the operating-system kernel. We extend the semantics accordingly in Sect. 4.

2.2 Fundamentals of our Microkernel

In this section, we sketch the features of VAMOS and explain the fundamental access-control mechanism that establishes the process roles "privileged process" and "device driver".

Our microkernel VAMOS performs the following tasks: (a) enforcement of a minimal access control, (b) process management, (c) memory management, (d) priority-based round-robin scheduling, (e) support for user-mode device drivers, and (f) inter-process communication (IPC). Processes can control these tasks via the kernel's application binary interface (ABI). Table 1 lists the kernel calls that constitute the ABI.

Most kernel calls are reserved for so-called *privileged processes*. Thus, only a privileged process can bring up new processes or kill existing ones, alter the memory consumption of processes, change their scheduling parameters, or control the registration of device drivers. Any process, however, might use the IPC mechanism. Thus, the privileging serves as a minimal access control. We presume that the privileged processes constitute the user-mode parts of the operating system and implement a more sophisticated access-control mechanism. Non-privileged processes may then communicate with the privileged processes in order to request kernel services on their behalf.

Table 1. Application binary interface of the VAMOS kernel

Kernel Call	**Description**
Access Control	
`set_privileged` [p]	add a process to the set of privileged processes
Process Management	
`process_create` [p]	create a new process from a memory image
`process_clone` [p]	copy an already existing process
`process_kill` [p]	kill a process
Memory Management	
`memory_add` [p]	increase the amount of virtual memory for a process
`memory_free` [p]	decrease the amount of virtual memory for a process
Scheduling Mechanism	
`chg_sched_params` [p]	change scheduling parameters
Device Driver Support	
`change_driver` [p]	(un)register a process as a driver for a set of devices
`enable_interrupts` [d]	re-enable a set of interrupts after their successful handling
`dev_read` [d] / `dev_write` [d]	communicate with a certain device
Inter-Process Communication	
`ipc_send`	send a message to another process
`ipc_receive`	receive a message from another process
`ipc_request`	send a message and immediately wait for a reply
`change_rights`	manipulate IPC rights
`read_kernel_info`	receive information from the kernel

[p] call is reserved for privileged processes [d] call is reserved for device drivers

When VAMOS boots, it launches one single process, the *init process*. This process is privileged and has to set up the required servers of the operating system, start and register the device drivers, and possibly run initial applications.

A *device driver* is a user process, which is designated for the communication with certain devices. Only if a process is registered as a driver for a particular device, it may place read or write requests from or to that device, respectively. Moreover, the device driver is notified of interrupts from that device.

2.3 Implementation Design

Fig. 1 depicts the implementation structure of VAMOS. The lowest software layer is called *communicating virtual machines* (CVM). This layer encapsulates all hardware-specific low-level functionality, which is possibly using inline assembly. CVM has two major tasks: memory virtualization and switching between different threads of execution. Hence, CVM includes a page-fault handler with a simple memory swapping facility [4]. Moreover, it exports an interface of so-called *primitives* for the access and manipulation of user machines to the hardware-independent part. We have thereby established a solid framework for microkernel construction.

Using this framework, we have implemented our microkernel VAMOS in C0 without extra portions of inline assembly. On every kernel entry, CVM preserves the old context, establishes a suitable C0 environment and calls the function `kdispatch` of VAMOS. For the manipulation of the user memory or registers, VAMOS may call the primitives of CVM. The return value of `kdispatch` determines, which user machine resumes when the kernel execution finishes.

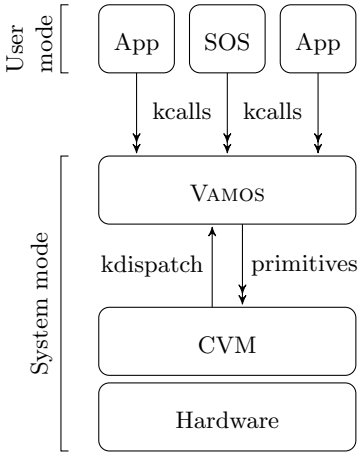

Fig. 1. Implementation scheme

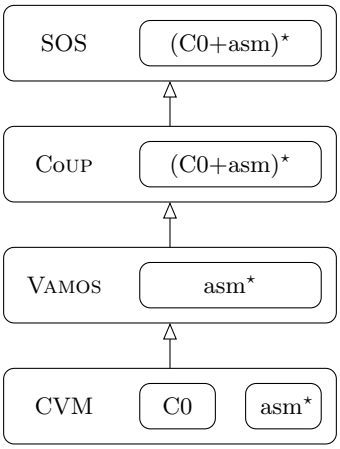

Fig. 2. Schematics of our model stack

While CVM and Vamos run in the privileged system mode of the processor, user machines run in the unprivileged user mode. In the figure, we labelled one user machine "SOS" for *simple operating system* and the others "App" as abbreviation for application. The SOS constitutes the highest layer of our operating system. It features an advanced rights management with different users, implements a sophisticated access control to kernel services like process creation and provides further services like file-system and network access. All user machines interact with the kernel via *kernel calls*. The special instruction `trap` causes an exception, which is handled in Vamos. Vamos can examine and alter the state of the user machine using CVM primitives, thus identifying the user machine's specific request and storing the kernel's corresponding response.

3 Model-Stack Overview

In analogy to the software stack, we have developed a model stack. For each software layer, this stack contains a specification that fully describes the observable behavior of this layer. The specification describes the full functionality of the software layer, on the one hand, and forms the foundation for the simulation proofs for the more abstract layers, on the other hand. We build the model stack as a means of verifying statements about the real system on a convenient, abstract layer. Such statements include safety and liveness properties.

Fig. 2 shows our model stack. The lowest model describes CVM. This model comprises a kernel machine, given in the C0 semantics, and a number of user machines with virtual memory, given in the assembly semantics. The kernel runs in system mode with disabled interrupts. The user machines, in contrast, may be interrupted after each assembly instruction, thus we model them on assembly level. We will not descend into the details of this model in the present publication because it has been described earlier [5].

We can instantiate the kernel of CVM with the Vamos implementation, i. e., the C0 program that implements (the hardware-independent part of) our microkernel is executed as CVM's kernel machine. The Vamos model specifies the behaviour of the resulting system. In short, this model establishes assembly processes that communicate with the kernel via a well-defined interface, the kernel ABI. A code-correctness proof for the Vamos implementation establishes the simulation theorem between the instantiated CVM model and the Vamos specification.

While the Vamos processes use the assembly semantics, the SOS and the applications are written in C0. The kernel ABI is encapsulated in a few library functions. Except for the library functions, the verification of these programs should certainly employ the more abstract C0 semantics. For this purpose, we designed the model *communicating user processes* (Coup). This model simulates Vamos but it abstracts from the scheduler and establishes C0 processes.

We need to abstract from the particular VAMOS scheduler because of the different granularity of the program semantics: The execution of a process might be suspended and resumed on every assembly instruction but a C0 statement usually contains several assembly instructions. The simulation proof has three parts: (1) The execution traces of COUP should contain all traces of VAMOS. (2) There is a confluent reordering of any execution trace such that there is no scheduling event during a single C0 statement. (3) The kernel library is correctly implemented.

We can instantiate one of the C0 processes with the implementation of SOS. The SOS model specifies the behaviour of the resulting system. The simulation between both models can be shown by the functional correctness of the SOS implementation.

4 Process Abstraction

In this section, we formalize the interface between the processes and the kernel. Both kernel models, the VAMOS specification and the COUP model, use this process abstraction in order to access and manipulate processes. Our abstraction is based on the observation that VAMOS interacts with processes only via a well-defined interface, the kernel ABI. Hence, we can encapsulate processes in a self-contained input-output automaton, thereby hiding the internal state and just exposing the generic interface.

We define a process as input-output automaton described by a tuple

$$(\mathcal{S}_{\text{proc}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \omega_{\text{proc}}, \mathit{vm\text{-}size}_{\text{proc}}, \mathit{init}_{\text{proc}}, \delta_{\text{proc}})$$

with state space $\mathcal{S}_{\text{proc}}$, input alphabet Σ_{proc}, output alphabet Ω_{proc}, output functions ω_{proc} and $\mathit{vm\text{-}size}_{\text{proc}}$, initialization function $\mathit{init}_{\text{proc}}$, and transition function δ_{proc}.

While the state space $\mathcal{S}_{\text{proc}}$ depends on the individual process abstraction, the interface between the kernel and the processes is naturally shared by all process abstractions. This interface is entirely defined by Σ_{proc} and Ω_{proc}.

The output alphabet Ω_{proc} enumerates all possible kernel calls. Additionally, we have to treat a few error cases. As the kernel calls are internally identified by a number, a process might specify an invalid number. This condition is represented by the special output value undefined_trap. Moreover, a process might generate exceptions like an arithmetic overflow or an illegal page fault. These exceptions are collectively represented as value runtime_error. Finally, the output ε denotes the intention to perform a local computation.

The input alphabet Σ_{proc} reflects all kernel-initiated changes of a process. These comprise all possible responses to kernel calls, on the one hand, and the demand to change the amount of virtual memory, on the other hand. While the

former are the synchronous reaction to a kernel call, the latter may be issued asynchronously at any stage of a process. In order to perform a local transition, we pass the input ε to the transition function δ_{proc}.

In addition to the usual functions $\delta_{\text{proc}} \in \Sigma_{\text{proc}} \times \mathcal{S}_{\text{proc}} \to \mathcal{S}_{\text{proc}}$ for transitions and $\omega_{\text{proc}} \in \mathcal{S}_{\text{proc}} \to \Omega_{\text{proc}}$ for the output, we use two more functions in our process abstraction. In order to compute the overall memory consumption of the process system, VAMOS needs to know the amount of virtual memory that is currently occupied by every process. The function $\mathit{vm\text{-}size}_{\text{proc}} \in \mathcal{S}_{\text{proc}} \to \mathbb{N}$ provides the necessary information. When a new process is created, VAMOS has to translate a representation of a given binary executable file into the corresponding, initial process state. We specify this translation in the function $\mathit{init}_{\text{proc}} \in \mathbb{N}^\star \to \mathcal{S}_{\text{proc}}$.

For the kernel specification, the generic abstraction of processes as self-contained input-output automata is rather a matter of taste than necessity. However, this generic process abstraction is a cornerstone of the CoUP model. Moreover, it is crucial for the simulation theorem between VAMOS and CoUP because both models mainly differ in the process abstraction. Hence, it proved to be beneficial that many definitions of the specification are parametrized over the process abstraction and can thus be reused in CoUP as well. Finally, the introduction of this parameter on the whole CoUP model is just a logical consequence because it paves the way for the integration of other language semantics.

In the following, we refine the generic abstraction with specific process models for assembly and C0.

Assembly Processes. We reuse the state space $\mathcal{S}_{\text{asm}}$ of the assembly semantics for our assembly processes. Based on this state space, we now define the output functions ω_{asm} and $\mathit{vm\text{-}size}_{\text{asm}}$, the transition function δ_{asm}, and the initialization function $\mathit{init}_{\text{asm}}$. The function $\mathit{vm\text{-}size}_{\text{asm}}$ simply returns the value of the component V of the current state: $\mathit{vm\text{-}size}_{\text{asm}}(s_{\text{asm}}) = s_{\text{asm}}.V$. However, the definitions for other functions are too complex to fully present them here. We constrain ourselves to an exemplary excerpt.

Fig. 3 depicts the formal definition of the output function ω_{asm} and the transition function δ_{asm} for the call `process_clone`. We assume that s_{asm} is the state of an assembly process. The predicate *trap* holds iff the current instruction is a trap, and the function *simm* extracts the sign-extended immediate constant from the current instruction. If there is a trap with immediate constant 2, the output function will return the pair of value process_clone and the content of register 11. Let us now assume that the kernel recognizes this output from the current process but the process is not privileged. In this case, the kernel signals this error condition by passing value err_unprivileged via the transition function to the current process. Then, the transition function updates the register 22 with the corresponding error code and increases the program counters.

$$trap(s_{\text{asm}}) \wedge simm(s_{\text{asm}}) = 2 \implies \omega_{\text{asm}}(s_{\text{asm}}) = (\mathsf{process\_clone}, s_{\text{asm}}.gpr(11))$$

$$\delta_{\text{asm}}(\mathsf{err\_unprivileged}, s_{\text{asm}}) = s_{\text{asm}} \left[\begin{array}{rcl} gpr(22) & := & -4 \\ pc & := & s_{\text{asm}}.pc + 4 \\ dpc & := & s_{\text{asm}}.dpc + 4 \end{array} \right]$$

Fig. 3. Formal definition of the output function and the transition function of assembly processes for the call `process_clone`

C0 Processes. In order to represent C0 processes, we define an automaton with the following signature:

$$(\mathcal{S}_{\text{C0}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \omega_{\text{C0}}, \mathit{vm\text{-}size}_{\text{C0}}, \mathit{init}_{\text{C0}}, \delta_{\text{C0}})$$

Any state $s_{\text{C0}} \in \mathcal{S}_{\text{C0}}$ comprises the static program environment as well as the dynamic program state of the C0 machine as described in Sect. 2.1. We store the program environment because processes might dynamically be created and killed, hence the program might change. The output functions ω_{C0} and $\mathit{vm\text{-}size}_{\text{C0}}$, the transition function δ_{C0}, and the initialization function $\mathit{init}_{\text{C0}}$ are defined in analogy to their assembly-process counterparts.

For example, the formal definition of the output function and the transition function for the call `process_clone` is given in Fig. 4. We assume that s_{C0} is the current state of a C0 process. The component $s_{\text{C0}}.\mathit{prog}$ denotes the remaining program of the process. We consider the first statement of the program. If this statement is a call to the function `vc_process_clone`, we define the output of ω_{C0} as the pair of process_clone and the value of the first argument to the function call. Let us now assume that the kernel recognizes this output from the current process, clones the given process, and computes the value hn_{new} as new identifier for the clone. It would then pass this value via the transition function to the current process, thus signalling the success of the clone operation. In this case, the transition function updates the memory $s_{\text{C0}}.\mathit{mem}$ at the address designated by the left-value e with value hn_{new} and removes the function call from the remaining program.

5 Formal Kernel Models

In the previous section, we described our process model. Now, we embed the processes into the two concurrent kernel models, the VAMOS specification, and the COUP model. The former specifies the exact behaviour of our microkernel with a particular scheduler. This model is used for code verification. The latter abstracts the scheduler and focusses on the interaction of the processes with the

$$s_{\mathrm{C0}}.prog = \text{"}e = \texttt{vc\_process\_clone}(e_0); r\text{"} \implies \omega_{\mathrm{C0}}(s_{\mathrm{C0}}) = (\mathsf{process\_clone}, \mathit{get\text{-}val}(s_{\mathrm{C0}}.mem, e_0))$$

$$s_{\mathrm{C0}}.prog = \text{"}e = \texttt{vc\_process\_clone}(e_0); r\text{"} \implies \delta_{\mathrm{C0}}((\mathsf{succ\_new\_process}, hn_{\mathrm{new}}), s_{\mathrm{C0}}) = s_{\mathrm{C0}}\begin{bmatrix} mem & := & \mathit{set\text{-}val}(s_{\mathrm{C0}}.mem, e, hn_{\mathrm{new}}) \\ prog & := & r \end{bmatrix}$$

Fig. 4. Formal definition of the output function and the transition function of C0 processes for the call `process_clone`

microkernel.We need this more abstract model in order to describe the reordering of interleaved sequences and to introduce C0 processes.

Both models are Moore machines. We describe the kernel specification with the tuple $(\mathcal{S}_{\mathrm{v}}, s^0_{\mathrm{v}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\mathrm{v}}, \delta_{\mathrm{v}})$ and the CoUP model with $(\mathcal{S}_{\mathrm{vc}}, s^0_{\mathrm{vc}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\mathrm{vc}}, \delta_{\mathrm{vc}})$. The state spaces $\mathcal{S}_{\mathrm{v}}$, $\mathcal{S}_{\mathrm{vc}}$ contain the initial states s^0_{v} and s^0_{vc}, respectively. For device communication, we use the input alphabet $\hat{\Sigma}$ and the output alphabet $\hat{\Omega}$. The functions ω_{v} and ω_{vc} determine the output in a current state. Finally, δ_{v} and δ_{vc} describe the transitions of the models. The notable difference between these machines regards the determinism: While the VAMOS specification is fully deterministic, CoUP features a non-determinism in its scheduling decisions. Consequently, the transition relation δ_{v} is functional while δ_{vc} is not.

Below, we introduce the different components of the models side by side.

Device Communication. Our kernel uses memory-mapped I/O for device communication. Hence, the output alphabet $\hat{\Omega}$ comprises read and write accesses to device addresses. The input alphabet $\hat{\Sigma}$ consists of interrupt lines and optionally incoming data. Hillebrand *et al.* [6] have described our device interface in detail.

State Spaces. A state $s_{\mathrm{v}} \in \mathcal{S}_{\mathrm{v}}$ comprises the following components:

- The partial function $s_{\mathrm{v}}.procs$ maps the process identifiers (PIDs) of the currently active processes to their assembly states $s_{\mathrm{asm}} \in \mathcal{S}_{\mathrm{asm}}$. For inactive processes, this function is undefined.
- Priorities are assigned to each PID of the active processes with the partial function $s_{\mathrm{v}}.priodb$.
- All other scheduling information is kept in the component $s_{\mathrm{v}}.schedds$.
- The partial function $s_{\mathrm{v}}.rightsdb$ maps PIDs to a data structure for the management of IPC rights and the set of privileged processes.
- Finally, the component $s_{\mathrm{v}}.devds$ contains data for device communication.

At first sight, the separation of priorities from the remaining scheduling data might surprise, here. The reason for this unintuitive partitioning is that CoUP abstracts from the particular scheduling algorithm but the priorities remain visible. We modularized the states and the corresponding transition functions in

order to share the definition of similar parts between the VAMOS specification, CoUP, and the SOS model.

A central part of the VAMOS specification are the scheduling data structures. The component $s_{\mathrm{v}}.\mathit{schedds}$ is divided into sub components. The current time $\mathit{time} \in \mathbb{N}$ is a counter for clock ticks. Process-specific scheduling information for active processes is collected in the partial function *procdb* that maps PIDs to a record of (a) the time slice *tsl*, (b) the amount of consumed time *ctsl*, and (c) the absolute timeout *to*. If a process is found to be computing when a timer interrupt raises, the component *ctsl* is increased until the process has finally run for *tsl* ticks. In this case, another process is scheduled. If a process calls the kernel for IPC and no partner is ready for communication, the absolute timeout *to* is computed from the current time and the relative timeout that has been specified with the call.

Moreover, the scheduler maintains different queues for scheduling. They are represented as finite sequences in the VAMOS specification. Namely, there is a ready queue $\mathit{ready}(\mathit{prio})$ of schedulable processes for each priority $\mathit{prio} \in \{0, 1, 2\}$. The processes that cannot currently be scheduled (because they are waiting for an IPC partner) are held in a queue named *wait*.

In VAMOS, the current process is the first process in the highest, non-empty ready queue. If all ready queues are empty, the current process is undefined. Formally, we define the function *cup* as:

$$\mathit{cup}(s_{\mathrm{v}}.\mathit{schedds}) = p \iff \exists i : s_{\mathrm{v}}.\mathit{schedds}.\mathit{ready}(i) = (p, \dots) \wedge \forall\, j > i : s_{\mathrm{v}}.\mathit{schedds}.\mathit{ready}(j) = ()$$

The corresponding CoUP state s_{vc} inherits most components of s_{v}. Only two components change: The process abstraction becomes a model parameter, i. e., component $s_{\mathrm{vc}}.\mathit{procs}$ of a CoUP state s_{vc} is a partial function from PIDs to a generic state space $\mathcal{S}_{\mathrm{proc}}$ for processes. Moreover, the scheduling data structures are replaced by a current-process indicator. We retain the current process in the state in order to compute the output from the current state. The output function ω_{vc} signals the demand for device communication. In order to determine this demand, we need to employ the output function ω_{proc} of the current process. Consequently, we fix this process beforehand instead of including transitions for all ready processes in the transition relation.

Transitions. A transition $\delta_{\mathrm{v}}(\hat{\sigma}, s_{\mathrm{v}})$ of the VAMOS specification under the device input $\hat{\sigma} \in \hat{\Sigma}$ has up to three phases:

1. If the current process $\mathit{cp} = \mathit{cup}(s_{\mathrm{v}}.\mathit{schedds})$ is defined, we consult its output $\omega_{\mathrm{proc}}(s_{\mathrm{v}}.\mathit{procs}(\mathit{cp}))$ and compute the response according to the current VAMOS state. For instance, if a process calls `process_clone`, we check for sufficient privileges and resources and choose the corresponding response

$\sigma \in \Sigma_{\text{proc}}$ for success or failure. With this response, we advance the current process: $s_{\text{v}}\,[procs(cp) := \delta_{\text{proc}}(\sigma, s_{\text{v}}.procs(cp))]$.
2. If the timer-interrupt line is raised, the scheduler increases the clock-tick counter $s_{\text{v}}.schedds.time$ and the consumed time $s_{\text{v}}.schedds.procdb(cp).ctsl$ of the current process. Moreover, the scheduler wakes up all processes p with elapsed timeouts.
3. Finally, VAMOS delivers interrupts to waiting drivers and saves the remaining interrupts for later delivery in $s_{\text{v}}.devds$.

The COUP transitions behave very similarly. However, a transition $s_{\text{vc}} \xleftrightarrow{\sigma} s'_{\text{vc}} \in \delta_{\text{vc}}$, obtains cp directly from $s_{\text{vc}}.cup$. Moreover, the only visible effect of the second phase is the wake-up of certain waiting processes. This effect is simulated non-deterministically and independently from the timer interrupt.

Reusability. As mentioned earlier, we worked hard to reuse as much definitions as possible in the different layers of abstraction. For example, in contrast to the COUP model, the kernel specification does not actually rely on the encapsulation of processes as self-contained input-output automata. Since VAMOS only considers assembly processes, it would be easier to directly manipulate the state of these processes. However, in order to literally reuse update functions and definitions, we invested the extra work and already introduced the input-output automata at this level of our model stack.

Consider the formal definition of the function *v_clone_updt_procs* in Fig. 5. This function describes the necessary updates in case of a successful call to `process_clone`. Without going into detail, depending on the relation between the calling process (p_{subj}) and the process that was cloned (p_{obj}), this function updates the state of the calling process, the state of the cloned process, and the state of the new process. The important thing about this definition is that the updates are presented in a process-abstraction independent manner. Rather than updating particular registers, as it would suffice for assembly processes, we feed the generic transition function δ_{proc} with generic input, e.g. $(\mathsf{succ\_new\_process}, hn_{\text{new}})$. Now, this generic interface between kernel and processes allows us to use *v_clone_updt_procs* in both models, i.e. VAMOS *and* COUP. Proceeding in a similar way with all kernel calls keeps the models clear and maintainable. Furthermore, tedious equivalence proofs between VAMOS and COUP are avoided.

6 Conclusion

Related Work. With the CLI stack [7], a pioneering approach started out for pervasive verification of a complete system with rigorous formalizations of specifications and proofs. Most notably, the simple kernel KIT [8] was developed

$$
\begin{array}{l}
v\_clone\_updt\_procs(procs, p_{\text{subj}}, p_{\text{obj}}, p_{\text{new}}, hn_{\text{new}}) = \\
\quad \lambda x. \\
\qquad \textbf{if } x = p_{\text{subj}} \textbf{ then } \delta_{\text{proc}}((\mathsf{succ\_new\_process}, hn_{\text{new}}), procs(p_{\text{subj}})) \\
\qquad \textbf{else if } x = p_{\text{new}} \wedge p_{\text{subj}} = p_{\text{obj}} \textbf{ then } \delta_{\text{proc}}((\mathsf{succ\_new\_process}, 0), procs(p_{\text{subj}})) \\
\qquad \textbf{else if } x = p_{\text{new}} \textbf{ then } procs(p_{\text{obj}}) \\
\qquad \textbf{else } procs(x)
\end{array}
$$

Fig. 5. Formal definition of the function *v_clone_updt_procs* that computes the component *procs* after a call to `process_clone`

and its machine code implementation was proven to be correct. However, this kernel is fairly simple compared to modern microkernels.

Many recent projects undertake verification efforts on modern microkernels. Among them are L4.verified [9], VFiasco [10], Eros [11], and the Flint project [12]. Though these projects may have achieved some advances, they all discard pervasiveness but focus on the verification of a single software layer. The Flint project has developed a verification framework for an assembly language and has formally proven the correctness for context-switching code. The other three projects have established semantics for C variants and have verified different properties on source code level. As far as we can see, inlined assembly portions are just postulated to be correct and solely described by their semantical effects. Moreover, these projects rely on compiler correctness.

Traditionally, concurrent systems are described by the *Calculus of Communicating Systems* [13] or as *Communicating Sequential Processes* (CSP) [14]. Both approaches, however, argue on a very abstract level and are mainly used to specify system requirements at an early stage in the system design. Basin, Olderog and Sevinç [15], for instance, describe the combination of CSP and Object-Z in order to specify and analyze security automata. Unfortunately, a link to the actual code is missing.

Discussion. We presented an approach to a pervasive, formal specification of a microkernel-based operating system. The formal models of our microkernel Vamos and the SOS occupy almost 400 kB and comprise about 8,000 lines of specification. The formalization took us about 48 person months. The different specification layers resemble essentially the implementation layers.

A recurring problem was the semantic gap between the high-level implementation language C0 and certain hardware details that were manipulated by the considered programs. Such details concerned hidden hardware components like devices, on the one hand, and the granularity of atomic operations on the hardware level, on the other hand. Both problems arose simultaneously when we argued about C0 processes: In pure C0, we cannot express kernel calls. Hence, we extended the original C0 semantics by a library of functions that implement the

kernel calls using inlined assembly. Moreover, user machines may be interrupted after the execution of any assembly instruction. Consequently, we had to justify that we may confluently shift all rescheduling events to statement boundaries before we could argue about C0 processes. In non-pervasive approaches, these challenges are likely to be skipped.

When we began to specify the different layers of our operating system, we started with independent models. As expected, many parts of the formal models overlapped because we specified the same operating system at several layers of abstraction. After a short period of evaluation for each abstraction layer, we coupled the models as tight as possible in order to minimize the proof efforts for the simulation theorems between adjacent models.

This ambivalent approach was very successful. In the very beginning, the specifications change very often, such that maintaining dependencies between them would be very tedious and distracting from the individual problems. As soon as the models stabilized, however, there was a common ground for a tight meshing. The synergies between the layers could easily be identified and utilized.

The key to shared definitions between the VAMOS specification and the COUP model was twofold: On the one hand, we unitized the state space in a way that permits easy abstraction between the different abstraction layers. On the other hand, we established the process abstraction already at the VAMOS specification. Both arrangements together enabled a very tight mesh, which could even be reused in the SOS specification.

Of course, this tight mesh of the models has drawbacks: The specification of an abstract layer depends massively on the underlying layers, which impedes the understanding of this model. Moreover, even the lower layers became more complicated, so for instance, when the process abstraction was already introduced in the VAMOS specification. Finally, several layers were affected whenever the abstraction at one level turned out to be wrong.

From our experience, however, we conclude that the advantages outweigh the drawbacks. We saved valuable time by reusing the definitions, which we otherwise had spent in distracting equivalence proofs. For instance, we succeeded with the step-wise simulation proof between the VAMOS specification and the COUP model within two months. We have proven this simulation for most kernel calls within days. Most of the time, however, we spend with the verification of IPC. This effort was caused by a considerably simpler modelling of IPC in COUP, which became possible after the scheduler was abstracted. In this particular case, the simpler modelling was desirable and hence, we invested the additional time for the proof. Independent models, however, would possibly have caused such an overhead for every kernel call.

In the code-correctness proof for VAMOS, there is only a minimal overhead (hours) induced by the use of the process abstraction in the specification. All in all, we invested about 18 person months to show the correctness of the IPC send

call. Including all auxiliary functions, the implementation of this call accounts for nearly half the code size and comprises the most complex parts. Again 18 person month had been spent on the functional verification of the file system and the hard disk driver code in the SOS. This code covers 20% of the SOS implementation.

Summary. The fundamental difference of our "pervasive" approach compared to "incremental" approaches that focus on a single layer at a time regards the design of specifications: At first, models are usually adapted to the actual verification goal. If it is the only goal, to show that the formal specification precisely describes the implementation, the specification tends to move closely towards the implementation. If a model is used as fundament of an underlying component without a proof, there is a likelihood that the model over-abstracts the actual implementation. At second, regarding one single layer at a time necessarily leads to self-contained, independent models. When combining those different layers later on, there is a tremendous proof effort necessary to establish simulation theorems between the adjacent layers. Our approach prevents this effort by the specification design.

References

1. Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Workshop on Systems Software Verification, to appear, Elsevier (2008)
2. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM. (September 2005) 2–11
3. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. STTT **8**(4-5) (2006) 411–430
4. Alkassar, E., Starostin, A., Schirmer, N.: Formal pervasive verification of a paging mechanism. In: TACAS. (2007)
5. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: TPHOLs. Volume 3603 of LNCS., Springer (2005) 1–16
6. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD, IEEE Computer Society (2005) 309–316
7. Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. J. Autom. Reasoning **5**(4) (December 1989) 411–428
8. Bevier, W.R.: Kit and the short stack. J. Autom. Reasoning **5**(4) (December 1989) 519–530
9. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: taking microkernels to the next level. Operating Systems Review (July 2007) 41(3)
10. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: the vfiasco project. In: Operating Systems Review, European workshop: beyond the PC, New York, NY, USA, ACM Press (2002) 165–169
11. Shapiro, J.S., Weber, S.: Verifying the EROS confinement mechanism. In: Symposium on Security and Privacy, IEEE Computer Society (2000) 166–176
12. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: TPHOLs, Lecture Notes in Computer Science (September 2007) 189–206
13. Milner, R.: A Calculus of Communicating Systems. Springer (1982)
14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
15. Basin, D.A., Olderog, E.R., Sevinç, P.E.: Specifying and analyzing security automata using CSP-OZ. In: ASIACCS, ACM (2007) 70–81