A Verification Approach for System-level Concurrent Programs*

Matthias Daum, Jan Dörrenbächer, Burkhart Wolff, and Mareike Schmidt

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

Abstract. Though the verification of operating systems is an active research field, a verification method is still missing that provides both, the proximity to practically used programming languages such as C *and* a realistic model of concurrency, i. e., a model that copes with the granularity of atomic operations actually used in a target machine.
Our approach serves as the foundation for the verification of concurrent programs in C0 – a C fragment enriched by kernel communication primitives – in a Hoare-Logic. C0 is compiled by a verified compiler into assembly code representing a cooperative concurrent transition system. For the latter, it is shown that it can actually be executed in a true concurrent way reflecting the C0 semantics.

1 Introduction

Industrial-strength software analysis and verification has advanced in recent years through the introduction of static analysis techniques, model checking as well as automated and interactive theorem proving. However, many techniques are working under restrictive assumptions that limit their applicability to complex (embedded) system software such as operating system kernels, low-level device drivers or microcontroller code.

In this paper, we present a theorem-proving based method that can cope with unbounded data-structures (buffers, stacks), in contrast to model-checking techniques limited to state spaces that have usually to be finite and small.

Based on Leinenbach and Petrova's [1] proven correct compiler from a C variant – called C0 – into assembly language, we provide the foundation of an abstract verification technique for concurrent programs on top of a particular microkernel. The main problem is to establish a relation between a true concurrent execution on a low-level machine and a cooperative concurrent model, which is suitable for verification. With *cooperative concurrent execution*, we refer to a conventional sequential execution except when basic communication primitives are called; after their call, a process may resume with a non-deterministically

* 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.

updated state (although the possible places where memory might change is actually very restricted in our model). By *true concurrent*, we mean an execution where a process may be arbitrarily interrupted and resumed.

In our paper, we label sequential models with the index $^{\text{seq}}$, cooperative concurrent models with $^{\text{cc}}$ and true concurrent ones with $^{\text{tc}}$. Fig. 1 presents the "grand picture" over the stack of languages; subsequently, we will briefly introduce to the underlying machines:

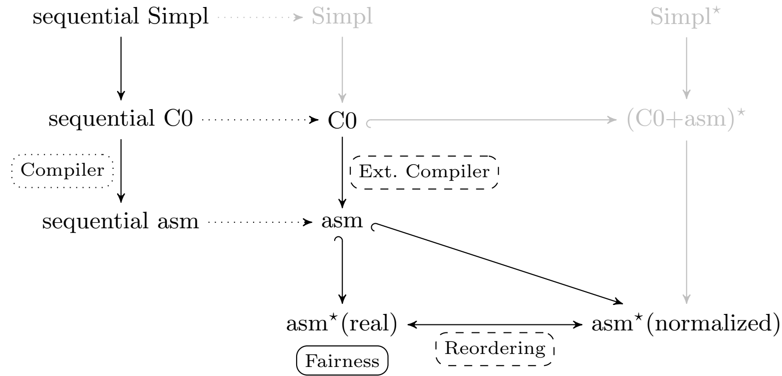

Fig. 1. Overview of the Language Stack

The core of this language stack is the small-step execution model $A^{\text{cc}}_{\text{C0}} = (\mathcal{S}_{\text{C0}}, \mathcal{S}^0_{\text{C0}}, \delta^{\text{cc}}_{\text{C0}})$ associated to "C0". The states $\mathcal{S}_{\text{C0}}$ of the automaton are the *configurations* of the small step semantics, i. e., pairs $(prog, (mem, \Gamma))$ of a C0-program *prog* yet to be executed, the program environment Γ, and the current memory *mem*. As usual, the initial states $\mathcal{S}^0_{\text{C0}}$ are a subset of $\mathcal{S}_{\text{C0}}$, and the transition relation $\delta^{\text{cc}}_{\text{C0}}$ is a binary relation over states. The language is a straight-forward extension of Leinenbach's "sequential C0" by kernel communication primitives. In this model, it is assumed that concurrency can *only* occur at these primitives, and that the result of actions of the process environment are only visible in form of non-deterministic changes in (a restricted) area of *mem*.

Associated to "asm" is the machine $A^{\text{cc}}_{\text{asm}}$, which executes the assembler operations like Leinenbach's "sequential asm" machine; however, $A^{\text{cc}}_{\text{asm}}$ is extended by kernel primitives related to operations of the microkernel VAMOS. This machine is more concrete that the abtract $A^{\text{cc}}_{\text{C0}}$ machine because it has no strict separation between code and data, and can therefore handle self-modifying code.

We associate "asm*(real)" to an operational VAMOS machine A^{tc}_{v}. This machine is a true concurrent model with a number of assembly processes and an explicit scheduler. There is an interface to external devices like a timer, a hard-disk or a console. Its transition relation is defined to simulate the $A^{\text{cc}}_{\text{asm}}$ machine.

Another operational kernel machine, $A^{\text{tc}}_{\text{vc}}$, is associated with "asm*(normalized)", which has no explicit scheduler but behaves otherwise like A^{tc}_{v}.

Schirmer [2] has proven a Hoare-Logic (Simpl) sound and complete wrt. to the sequential $A^{\text{cc}}_{\text{C0}}$ machine; furthermore, his work provides a whole verification environment comprising a verification condition generator as a means for effective verification of C0 programs. Albeit his work could be easily adapted to our language stack, we will not discuss this further and rather focus on the foundation for its use in a concurrent setting.

As main contributions of this paper, we provide:

- the extension of the sequential C0 semantics and the sequential assembly semantics with suitable communication primitives;
- an extended compiler correctness theorem: the translation of C0 (including the kernel primities) to assembly preserves the operational semantics,
- a formal proof in Isabelle/HOL that the operational kernel machine A^{tc}_{v} provides fair scheduling between assembly processes,
- a reordering theorem allowing for the reinterpretation of execution traces of the A^{tc}_{v} machine in terms of $A^{\text{cc}}_{\text{asm}}$.

Thus, we provide a method that is on the one hand concrete enough for the verification of user-mode device drivers and similar operating-system components. On the other hand, we can indeed reason in a sequential, process-local style via a Hoare-Logics over concurrent programs. However, this reasoning can only establish partial correctness of process-systems; in the future work section, we will therefore discuss the implications of our work for machines such as $(C0{+}asm)*$, i. e., concurrent thread systems, and its abstraction to Hoare-Logics $Simpl^\star$ enabling the proof of system global liveness properties.

2 Background

2.1 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 of the kernel calls are reserved for so-called *privileged processes*. Thus, only a privileged process can bring up new processes or kill existing ones; it is able to control the memory consumption of a process, can change its scheduling parameters or alter the registration of device drivers.

However, any process might use the IPC mechanism. Thus, the concept of privileged processes serves as a minimalistic access-control mechanism. We presume that these processes constitute the user-mode parts of the operating system

and implement a more sophisticated access control. Non-privileged processes may then use IPC in order to request the kernel services indirectly from the privileged processes.

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 potentially 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.

2.2 On a Correct Compiler

In this section, we summarize work of Leinenbach and Petrova [1,3]. 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 $x\,[\mathit{this} := q]$.

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

The Language C0. ANSI C has a complex and highly underspecified semantics. However, low-level kernel programs such as drivers explicitly *use* properties of a particular compiler on a target hardware, for example, its little-endianness or a particular atomicity of assembly operations. They can therefore not be verified according to the too vague ANSI C semantics. In our approach, we constrained ourselves to the C-like imperative language *C0*, which has sufficient features to implement low-level software, but which is interpreted by a more concrete semantics. C0's most important limitations compared to ANSI C are:

- expressions must be free of side effects and do not contain function calls,
- there are no implicit type conversions, especially not from arrays to pointers,
- pointers are strongly typed and must not point to functions or stack variables (i. e., there are neither void pointers nor pointer arithmetic), and
- low-level data types (like unions and bit fields) and control-flow statements (like switch and goto) are not supported.

Syntax. C0 supports fundamental types, aggregate types and pointers. The first category comprises Booleans, 8-bit-wide characters, as well as signed and unsigned 32-bit integers. Aggregate types in C0 are arrays and structures. Pointers may point to all types of data but not to functions.

Expressions are variable names and literals. Moreover, if e and i are expressions and n is a component name, array access $e[i]$, access to structure components $e.n$, dereferencing $*e$, and the "address-of" operation $\&e$ are expressions. Additionally, C0 supports the usual unary and binary operators.

Finally, C0 supports statements for assignments, dynamic memory allocation, sequential composition, conditional and repeated execution, inline assembly, function calls and returns from functions.

Small-step Semantics. For the lack of space, we can only glance at the 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 symbol table is a list of pairs of variable names and types. The type-name environment maps type names to types. The function table maps function names to functions, which are represented by a tuple consisting of (a) a symbol table for the function's parameters, (b) a symbol table for the stack variables, (c) the function's return type, and (d) a statement representing the function body.

A C0 state s_{C0} in execution is composed of

- the remaining program $s_{\mathrm{C0}}.prog$,
- the current state of the program variables $s_{\mathrm{C0}}.mem$, and
- the program environment (which remains constant in the sequential setting and may therefore be omitted there).

In the following sections, we assume an evaluation function *get-val* for the lookup, 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 $\textit{get-val}(s_{\mathrm{C0}}, e)$. If we update the left-value l in state s_{C0} with some expression u, we denote the resulting configuration by $\textit{set-val}(s_{\mathrm{C0}}, l, u)$.

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

The Target Assembly Language. The assembly semantics was developed for the DLX processor VAMP [4]. It abstracts from the paging mechanism of the processor and employs a linear memory model. Assembly states s_{asm} consist of the following components:

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

We denote the state space of the assembly semantics by $\mathcal{S}_{\mathrm{asm}}$. Assembly computation is modeled by the partial function $\delta_{\mathrm{asm}}^{\mathrm{seq}} \in \mathcal{S}_{\mathrm{asm}} \rightharpoonup \mathcal{S}_{\mathrm{asm}}$. Note that the effects of exceptions like illegal page faults cannot be fully determined from the assembly-machine state. In that case, $\delta_{\mathrm{asm}}^{\mathrm{seq}}$ 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. 3.

Compiler Correctness. We establish compiler correctness via a simulation relation. This relation employs an allocation function *alloc* that maps the current variables to memory locations. Based on the allocation function, we define a simulation relation $consis(alloc)(s_{\mathrm{C0}}, s_{\mathrm{asm}})$, which relates values of variables, pointers and the remaining program in a C0 state s_{C0} to their corresponding memory regions and the value of the program counter in an assembly state s_{asm}.

Compiler correctness states a stepwise simulation. If a C0 state simulates an assembly state, the simulation relation can again be established after a C0 step and a finite sequence of assembly steps. However, this statement is too general with respect to resource limitations and type correctness. Hence, we formulate:

Theorem 1 (Compiler Correctness). *Assuming that s_{C0} represents a well-typed C0 state, there is no runtime error in the next step, and there are sufficient resources, the following statement holds:*

$$consis(alloc)(s_{\mathrm{C0}}, s_{\mathrm{asm}}) \implies \exists n, alloc' : consis(alloc')(\delta_{\mathrm{C0}}^{\mathrm{seq}}(s_{\mathrm{C0}}), (\delta_{\mathrm{asm}}^{\mathrm{seq}})^n(s_{\mathrm{asm}}))$$

where $_^n$ is the n-fold function composition. Leinenbach and Petrova have formally shown this theorem in Isabelle/HOL.

3 Process Models

In this section, we formalize the interface between the processes and the kernel. Our formalization is based on the observation that VAMOS interacts with

processes only via a well-defined interface, which is the kernel ABI. Hence, we can encapsulate processes in a self-contained input-output automaton, thereby hiding the internal state and exposing only the generic interface. We use this encapsulation in order to abstract from assembly processes as they are found in the operational kernel models to the more abstract C0 processes. Our formalization refines of the cooperative concurrent models A^{cc} from the introduction. In order to precisely model the interaction with the kernel, we refined the transition relation δ^{cc} to a partial function parametrized over an input and introduced output functions.

We define a process A_{proc} as an input-output automaton described by a tuple

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

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

While the state space $\mathcal{S}_{\mathrm{proc}}$ depends on the individual process model, the interface between the kernel and the processes is naturally shared by all process models. 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 by the value runtime_error. Finally, the output ε denotes the intention to perform a local computation as the next step.

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 first mentioned inputs are the synchronous reaction to a kernel call, demands for memory changes might hit a process at any stage. In order to perform a local transition, we pass the input ε to the transition function δ_{proc}.

For the sake of memory management, VAMOS needs to know the amount of virtual memory that is currently occupied by a process. The function $\mathit{vm\text{-}size}_{\mathrm{proc}}$ provides the necessary information. When a new process is created, VAMOS has to transform a representation of the binary executable file into the inital process state. We encapsulate this transformation in the function $\mathit{init}_{\mathrm{proc}}$.

Below, we refine this generic interface with a specific interpretation for assembly processes and C0 processes and state an extended compiler correctness theorem that relate these process models.

Assembly Processes. We reuse the state space of the assembly semantics for our assembly processes. Based on this state space, we now define the output functions ω_{asm} and $\mathit{vm\text{-}size}_{\mathrm{asm}}$, the transition function δ_{asm}, and the initialization function $\mathit{init}_{\mathrm{asm}}$. The function $\mathit{vm\text{-}size}_{\mathrm{asm}}$ looks up the component V of the

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

$$\delta_{\mathrm{asm}}(\mathsf{err_unprivileged}, s_{\mathrm{asm}}) = s_{\mathrm{asm}} \begin{bmatrix} gpr(22) := -4 \\ pc := s_{\mathrm{asm}}.pc + 4 \\ dpc := s_{\mathrm{asm}}.dpc + 4 \end{bmatrix}$$

Fig. 2. Formal definition of output and transition function of assembly processes for the call `process_clone`

current state: $vm\text{-}size_{\mathrm{asm}}(s_{\mathrm{asm}}) = s_{\mathrm{asm}}.V$. The other functions are much more complicated and we cannot fully present them here.

However, Fig. 2 depicts an examplary excerpt from 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 process_clone and the value of register 11. Let us now assume that the kernel recognizes this output from the current process but the process is not privileged. The kernel then signals this error condition by passing the value err_unprivileged on to the current process via the transition function. In this case, the transition function updates the result register 22 with the corresponding error code and increases the program counters.

C0 Processes. Our user programs are implemented in C0. However, the pure C0 semantics cannot generate traps for the communication with the kernel. Hence, we extend the original C0 semantics with a special kernel library. This library comprises functions that use inline assembly code to implement the kernel calls. For example, Fig. 3 shows the implementation of function `vc_process_clone`. It loads the parameter `hn` (identifying the process to clone) into register 11, performs a trap passing constant 2, and returns the value of register 22.

Recall that the sequential C0 semantics implicitly assumes sufficient memory, which is not appropriate for C0 processes. Hence, we extend the state s_{C0} of the

```
int vc_process_clone(unsigned int hn) {
  int result;
  asm {  lw(r11, r30, asm_offset(hn));
         trap(2);
         sw(r22, r30, asm_offset(result));
       };
  return result;
}
```

Fig. 3. Implementation of function `vc_process_clone` from the kernel library

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

$$s_{\mathrm{C0}}.prog = \text{"}e = \texttt{vc_process_clone}(e_0); r\text{"}$$
$$\implies \delta_{\mathrm{C0}}(\mathsf{err_unprivileged}, s_{\mathrm{C0}}) = s_{\mathrm{C0}} \begin{bmatrix} \text{mem} := \mathit{set\text{-}val}(s_{\mathrm{C0}}.mem, e, -4) \\ \text{prog} := r \end{bmatrix}$$

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

C0 semantics by the memory size $s_{\mathrm{C0}}.msize$ of the process and do not define the partial transition function δ_{C0} for insufficient memory. We denote the set containing all states of C0 processes by $\mathcal{S}_{\mathrm{C0}}$. Based on this state space, we define the functions ω_{C0}, $\mathit{vm\text{-}size}_{\mathrm{C0}}$, δ_{C0}, and $\mathit{init}_{\mathrm{C0}}$ in analogy to their assembly-process counterparts. Function $\mathit{vm\text{-}size}_{\mathrm{C0}}$ looks up the component $s_{\mathrm{C0}}.msize$. For the other functions, we chose as prototypical example the formal definition for the call `process_clone` in Fig. 4.

For the C0 state s_{C0}, we consider the first statement of the program $s_{\mathrm{C0}}.prog$. If that is a function call to `vc_process_clone`, we define the output of ω_{C0} as a pair of process_clone and the function argument's value. Let us now assume that the kernel recognizes this output from the current process but the process is not privileged. The kernel then signals this error condition by passing the value err_unprivileged on to the current process via the transition function. In this case, the transition function updates the memory $s_{\mathrm{C0}}.mem$ at the address where the left-value e is stored with the corresponding error code and removes the function call from the remaining program.

Compilation. After having learned of the two process models, we now examine the compilation process and correlate both models. Our user programs are purely implemented in C0 but may contain function calls to the kernel library. For simplicity, we link on source-code level, i. e., the sources of the user program and of the kernel library are merged before compilation. Hence, the resulting C0 programs contain ordinary C0 statements on the one hand and library functions with inline assembly on the other hand.

Theorem 2 (Extended Compiler Correctness). *C0 processes simulate assembly processes.*

Proof Sktech. For ordinary C0 statements, we employ compiler correctness (Theorem 1). The correctness of the kernel library, however, can only be proven function by function on assembly level.

Fig. 5 shows the case of a library function which is called in some C0 state s_{C0}^n. From the compiler correctness theorem, we know that there exists a corresponding assembly state s_{asm}^b that satisfies the simulation relation $\mathit{consis}(\mathit{alloc})$. Now, we execute the inline assembly code starting in one such arbitrary, fixed s_{asm}^b. When the code reaches the trap instruction in state s_{asm}^c, the assembly

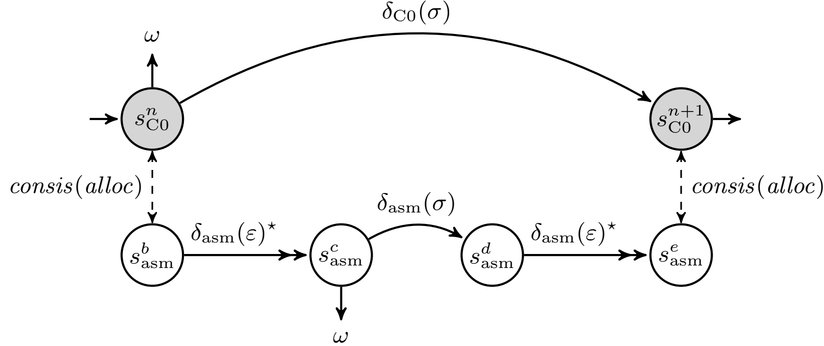

Fig. 5. Verification scheme for inline assembly portions in the kernel library

process has to signal the same output ω to the kernel as the C0 process does in s^n_{C0}. At this stage, the transition function δ_{asm} uses an input σ from the kernel to proceed. After further internal steps, we arrive at the end of the inline assembly portion in a state s^e_{asm}. We compute the corresponding C0 state $s^{n+1}_{\text{C0}} = \delta_{\text{C0}}(\sigma, s^n_{\text{C0}})$. Our library function is correctly implemented iff the pair $(s^e_{\text{asm}}, s^{n+1}_{\text{C0}})$ is in the simulation relation $consis(alloc)$. $\square$

A similar problem was formally proven with Isabelle by Starostin and Tsyban [5].

4 The True Concurrent Machines

In the previous section, we explained our process model. Now, we embed the processes into two true concurrent models, the Vamos specification A^{tc}_{v}, and the *Communicating User Processes* (Coup), formally described by $A^{\text{tc}}_{\text{vc}}$. 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 microkernel. We need this more abstract model in order to describe the reordering of interleaved sequences.

Both models are Moore machines. We describe them with the following tuples: $A^{\text{tc}}_{\text{v}} = (\mathcal{S}_{\text{v}}, s^0_{\text{v}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\text{v}}, \delta_{\text{v}})$ and $A^{\text{tc}}_{\text{vc}} = (\mathcal{S}_{\text{vc}}, s^0_{\text{vc}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\text{vc}}, \delta_{\text{vc}})$, respectively. The state spaces $\mathcal{S}_{\text{v}}$, $\mathcal{S}_{\text{vc}}$ contain the initial state 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 from 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 A^{tc}_{v} is fully deterministic, $A^{\text{tc}}_{\text{vc}}$ 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 and present a simulation theorem with regard to an abstraction function $abs \in \mathcal{S}_{\text{v}} \to \mathcal{S}_{\text{vc}}$. Though we abstracted in $A^{\text{tc}}_{\text{vc}}$ from the particular scheduler policy of Vamos,

we would like to preserve the property of fairness. For an arbitrary scheduler, we cannot encode this property into the transition relation. Hence, we formulate the property over an infinite sequence of transitions and show that $A_{\mathrm{v}}^{\mathrm{tc}}$ indeed fulfils this requirement. Finally, we argue on the possibility to reorder interleaved sequences while preserving the same system behaviour.

Device Communication. Our kernel uses a 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 $s_{\mathrm{v}}.\mathit{procs}$ 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 $s_{\mathrm{v}}.\mathit{priodb}$.
- All other scheduling information is kept in the component $s_{\mathrm{v}}.\mathit{schedds}$.
- The partial function $s_{\mathrm{v}}.\mathit{rightsdb}$ maps PIDs to a datastructure for the management of privileged processes and IPC rights.
- Finally, the component $s_{\mathrm{v}}.\mathit{devds}$ contains data for device communication.

The corresponding state $s_{\mathrm{vc}} = \mathit{abs}(s_{\mathrm{v}})$ inherits all components of s_{v} except for $s_{\mathrm{v}}.\mathit{schedds}$, which is replaced by a current-process indicator. We retain the current process in s_{vc} 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_{asm} of the current process. Consequently, we fix this process beforehand instead of including transitions for all ready processes in the transition relation.

We take a closer look at the scheduling data structures because they concern us in the following sections. The component $s_{\mathrm{v}}.\mathit{schedds}$ itself can be subdivided into components. The current time $\mathit{time} \in \mathbb{N}$ is a counter for the 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 timeslice *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 that case, another process will be 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 is specified with the call.

Moreover, the scheduler maintains different queues for scheduling. They are represented as finite sequences in $A_{\mathrm{v}}^{\mathrm{tc}}$. Namely, there is a queue $\mathit{ready}(\mathit{prio})$ of schedulable processes for each priority $\mathit{prio} \in \{0, 1, 2\}$. Processes waiting for an IPC partner are enqueued in the sequence *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:

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

Transitions. A transition $\delta_{\mathrm{v}}(\sigma, s_{\mathrm{v}})$ under device input σ 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 $\omega_{\mathrm{asm}}(s_{\mathrm{v}}.\mathit{procs}(\mathit{cp}))$ 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 response σ for success or failure accordingly. With this response, we advance the current process: $s_{\mathrm{v}}[\mathit{procs}(\mathit{cp}) := \delta_{\mathrm{asm}}(\sigma, s_{\mathrm{v}}.\mathit{procs}(\mathit{cp}))]$.
2. If the timer-interrupt line is raised, the scheduler increases the clock-tick counter $s_{\mathrm{v}}.\mathit{schedds}.\mathit{time}$ and the consumed time $s_{\mathrm{v}}.\mathit{schedds}.\mathit{procdb}(\mathit{cp}).\mathit{ctsl}$ of the current process. Moreover, the scheduler wakes up all processes p with elapsed timeouts, i. e., where the absolute timeout $s_{\mathrm{v}}.\mathit{schedds}.\mathit{procdb}(p).\mathit{to}$ is less than or equal to the current time $s_{\mathrm{v}}.\mathit{schedds}.\mathit{time}$.
3. Finally, VAMOS delivers interrupts to waiting drivers and saves the remaining interrupts for later delivery in $s_{\mathrm{v}}.\mathit{devds}$.

Respectively the same holds for every transition $s_{\mathrm{vc}} \overset{\sigma}{\longleftrightarrow} s'_{\mathrm{vc}} \in \delta_{\mathrm{vc}}$, except that we obtain *cp* directly from $s_{\mathrm{vc}}.\mathit{cup}$. Moreover, the only visible effect of a timer interrupt is the wake-up of processes with an elapsed timeout. This effect is simulated non-deterministically and independently from the timer interrupt.

Simulation Theorem. We would like to show that our more abstract model $A^{\mathrm{tc}}_{\mathrm{vc}}$ simulates the kernel specification $A^{\mathrm{tc}}_{\mathrm{v}}$. We formulate this fact over an infinite input sequence $\mathit{inputs} \in \mathbb{N} \to \hat{\Sigma}$ that maps a step number to a particular device input.

Theorem 3 (Equivalence of Transition Sequences). *The initial states are equivalent, i. e., $\mathit{abs}(s^0_{\mathrm{v}}) = s^0_{\mathrm{vc}}$, and their equivalence is preserved for all device inputs σ after each transition, i. e.,*

$$\mathit{abs}(s_{\mathrm{v}}) \overset{\sigma}{\longleftrightarrow} \mathit{abs}(\delta_{\mathrm{v}}(\sigma, s_{\mathrm{v}}))$$

Proof Insights. We proved this statement formally in Isabelle/HOL. As we could just glance at the transition relations in this paper, we can only summarize a few insights from our proof. All in all, the verification was straightforward because we reused the infrastructure of $A^{\mathrm{tc}}_{\mathrm{v}}$ to a large extent in $A^{\mathrm{tc}}_{\mathrm{vc}}$. The biggest difficulty we faced was in the verification of IPC because of a considerably simpler modelling in $A^{\mathrm{tc}}_{\mathrm{vc}}$, which became possible after the scheduler was abstracted. □

Fairness. We formulate fairness over infinite transition sequences, which we represent as a tuple consisting of two functions *states* and *inputs* that map the step number to the state and the input, respectively. We specify fairness as liveness for the processes and do not quantify the progress of the processes with respect to time slices. For an arbitrary, fixed process *pid*, we assume that it has the maximum priority infinitely often while the timer interrupt is active. A transition sequence preserves fairness if eventually there is a state where this process (a) is not active, (b) has a changed priority, (c) is starving in an IPC operation with an infinite timeout, or (d) is advancing.

Formally, we define:

$$
\begin{aligned}
&\mathit{isFair}(\mathit{inputs}, \mathit{states}) \equiv \forall \mathit{pid}\ k : \\
&\quad \forall n \geq k : \exists m \geq n : \mathit{maxprio}(\mathit{states}(m), \mathit{pid}) \wedge \mathit{isTimer}(\mathit{inputs}(m)) \\
&\quad \implies \exists l \geq k : \neg \mathit{defined}(\mathit{states}(l).\mathit{procs}(\mathit{pid})) \vee \\
&\qquad\qquad\qquad \mathit{states}(l+1).\mathit{priodb}(\mathit{pid}) \neq \mathit{states}(l).\mathit{priodb}(\mathit{pid}) \vee \\
&\qquad\qquad\qquad \mathit{starving_infinite_ipc}(\mathit{states}, l, \mathit{pid}) \vee \\
&\qquad\qquad\qquad \mathit{progress}\,(\mathit{states}(l).\mathit{procs}(\mathit{pid}), \mathit{states}(l+1).\mathit{procs}(\mathit{pid}))
\end{aligned}
$$

A ready process p has the current maximum priority if there is no process in a higher priorized ready queue, i. e., $\forall j > s.\mathit{priodb}(p) : s.\mathit{ready}(j) = ()$. As there are no queues in $A_{\mathrm{vc}}^{\mathrm{tc}}$ anymore, we rely on the fact that the current process has the maximum priority, i. e., $\mathit{maxprio}(s, p) \iff s.\mathit{priodb}(p) = s.\mathit{priodb}(s.\mathit{cup})$. The predicate $\mathit{starving_infinite_ipc}(\mathit{states}, l, p)$ examines a sequence of states *states* and holds iff the process p is pending in an IPC operation with an infinite timeout since step l. Finally, the predicate $\mathit{progress}(p, q)$ holds iff q can be produced by one or more transitions starting at p.

Theorem 4 (Fairness). *The* VAMOS *scheduler is fair, i. e., with the set* $\mathcal{R}_{\mathrm{v}}$ *containing the infinite transition sequences of* $A_{\mathrm{v}}^{\mathrm{tc}}$*, we can state*

$$\forall\,(\mathit{states}, \mathit{inputs}) \in \mathcal{R}_{\mathrm{v}} : \mathit{isFair}(\mathit{inputs}, \mathit{abs} \circ \mathit{states})$$

Proof Sketch. An arbitrary, fixed process *pid* can either be inactive, waiting, or ready. In the first case, Condition (a) of the predicate *isFair* holds immediately. If a process is waiting, it has called the kernel for an IPC operation and no partner is ready for it. In this case, there will eventually be a partner issuing a kernel call for IPC, the operation will time out, or the process is starving in an infinite IPC. The latter case corresponds to Condition (c), the former two imply progress (Condition (d)) because the kernel returns a result to the process.

Finally, there remains the case where the process is currently ready. Then, the process resides in the ready queue that corresponds to its priority. The process will be dequeued only if it becomes inactive, or if its priority changes (Conditions (a) and (b)). Assuming that the process has the current maximum priority infinitely often while the timer interrupt is active, the process will move forward in the ready queue until it is the first one. We know this fact because the scheduler will charge the current process if the timer interrupt is active and timeslices are bound by a fixed value. Hence, the process *pid* will eventually

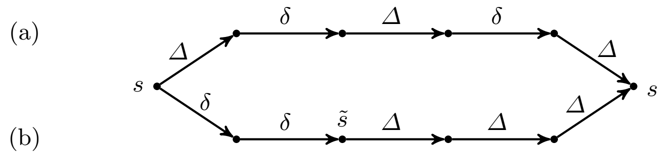

Fig. 6. Reordering transitions of the global system

be the first in its ready queue and thus, when it eventually has the maximum priority, it is the current process.

Usually, the current process advances immediately. Most notably, our implementation guarantees liveness, i. e., a started user process performs at least one step between two subsequent kernel entries. Still, there might be no immediate progress if the current process calls the kernel for an IPC operation. In this case, however, the kernel will enqueue the process in the wait queue, and we have shown fairness for all processes in this queue.

This proof has been formally developed in Isabelle/HOL. □

Shifting Scheduling Decisions. We have introduced $A^{\mathrm{tc}}_{\mathrm{vc}}$ as a means to reinterpret execution traces of the $A^{\mathrm{tc}}_{\mathrm{v}}$ machine such that rescheduling only takes place when the current process s_{asm} is about to execute a kernel primitive, i. e., $\omega_{\mathrm{asm}}(s_{\mathrm{asm}}) \neq \varepsilon$. This reinterpretation is possible if a process cannot observe that it was interrupted and resumed except at a kernel call. If processes do not observe scheduling decisions, the whole process system behaves confluently. We formulate:

Theorem 5 (Reordering). *Scheduling decisions of a process pid are confluent between two adjacent kernel calls of this process.*

Proof Sketch. For this proof, we can employ trace theory. In short, transitions can be commuted if they are independent of each other. Fig. 6 shows (a) an interleaved sequence of transitions between two states s and s' of the global system together with (b) a reordered sequence with a cooperative concurrent segment between s and $\tilde{s}$. In the figure, we have marked all transitions that do not affect the process *pid* by Δ and the local assembly steps of the process *pid* by δ. These transitions do not interfere with each other and can thus be commuted.

Non-commutable transitions, called *synchronization points*, affect simultaneously the kernel data structures and the considered process *pid*. The most prominent representative is a kernel call of *pid*. However, this is not the only one: The kernel may asynchronously change the memory size of processes. Strictly speaking, these transitions break our non-interference assumption and thus inhibit confluent reordering. However, we know from the kernel specification that the memory size is only changed if one of the privileged processes requests it. In practice, there should be a protocol between the processes ensuring that a process is always on a synchronization point if its memory is changed. Hence,

we can establish the reordering theorem if and only if the privileged processes request to change the memory size of process *pid* only at synchronization points or not at all.

As mentioned earlier, the set of privileged processes is usually very small. In a system with a single privileged process, the memory size of this process cannot be changed asynchronously. Hence, we may reason about this process in a cooperative concurrent fashion and show that it will not change the size of other processes unless they have requested it via IPC – and hence have reached a synchronization point. With this knowledge, we may reason about all other processes in a cooperative concurrent fashion. □

5 Conclusion and Future Work

Related Work. Bevier [7] was a precursor in operating-system verification. However, he verified a fairly simple kernel in comparison to modern microkernels.

Many recent projects undertake verification efforts on modern microkernels. Among them are VFiasco [8] and its successor Robin [9], L4.verified [10], Eros [11] and its successor Coyotos [12], as well as the Flint project [13]. Though these projects may have achieved some advances, they all focus on microkernel verification but do not ascend towards the verification of user programs.

Recently, Hobor *et al.* [14] have proposed a verification method for concurrent programming languages. The group assumes a fairly abstract language model and extends it with threads that communicate using locks while competing for a shared memory. Besides the differences in the communication model, this work is complementary to ours because Hobor *et al.* just assume a cooperative concurrent environment.

Achievements. Though kernel verification is often motivated as a foundation, the verification of operating-system components running on a kernel remains an open problem. So far, existing approaches only *postulate* a cooperative concurrent environment with fair scheduling.

In this paper, we bridge the gap between these cooperative concurrent models and a realistic, true concurrent execution machine. Thus, we provided the foundation for the verification of concurrent C0 programs. In particular, we extended a sequential C0 semantics and a sequential assembly semantics with kernel primitives, extended an existing compiler correctness theorem for the sequential semantics accordingly, formally proved that our microkernel is fair, and showed under which circumstances execution traces in the true concurrent system can be reordered to a cooperative concurrent ones. We have formally developed the proofs for Theorems 3 and 4 in Isabelle/HOL within ten person months.[1]

Future Work. Besides the formalization of Theorems 5 and 2 in Isabelle and the technical integration of the kernel primitives into Simpl, we see the following

[1] We are happy to provide the corresponding Isabelle theories upon e-mail request.

important extension of our work: So far, reasoning in terms of thread-local C0 programs is inherently only possible for partial correctness. In a thread-local view of the communication primitives like `ipc_receive`, it can not always be inferred if a process can actually continue execution or will get stucked because there is no communication partner. In order to establish total correctness for processes (what the Simpl framework potentially can), it is necessary to consider the global system state in a suitably extended Hoare-Logic Simpl* allowing us to reason about the system of processes as such.

Acknowledgements. We thank Sarah Hoffmann, Norbert Schirmer, and Irena Dotcheva for reviewing, constructive criticism and helpful suggestions.

References

1. Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Systems Software Verification, Elsevier (2008) to appear
2. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, TU Munich (2006)
3. Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM, IEEE Computer Society (2005) 2–12
4. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. STTT **8** (2006) 411–430
5. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Systems Software Verification, Elsevier (2008) to appear
6. Hillebrand, M.A., In der Rieden, T., Paul, W.J.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD, IEEE (2005) 309–316
7. Bevier, W.R.: Kit and the short stack. J. Autom. Reasoning **5** (1989) 519–530
8. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: the VFiasco project. In: ACM SIGOPS European Workshop, ACM (2002) 165–169
9. Tews, H.: Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. In: C/C++ Verification Workshop, technical report ICIS–R07015, Radboud University Nijmegen (2007) 59–68
10. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: taking microkernels to the next level. Operating Systems Review **41** (2007) 3–11
11. Shapiro, J.S., Weber, S.: Verifying the EROS confinement mechanism. In: IEEE Symposium on Security and Privacy. (2000) 166–176
12. Shapiro, J., Doerrie, M.S., Northup, E., Sridhar, S., Miller, M.: Towards a verified, general-purpose operating system kernel. In: FM Workshop on OS Verification. Technical Report 0401005T-1, National ICT Australia (2004) 1–19
13. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: TPHOLs, Springer (2007) 189–206
14. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: ESOP. Volume 4960 of LNCS., Springer (2008) 353–367