

Modelling User Programs on top of a Microkernel*

Matthias Daum

Saarland University, Germany
md11@wjpserver.cs.uni-sb.de

Abstract. Many verification techniques for concurrent systems only postulate properties of the underlying system such as fair scheduling. In my work, I establish a concurrent model that describes user programs running on top of a particular microkernel and prove that all assumptions of this model indeed hold for the kernel specification.

1 Introduction

Industrial-strength software verification has recently advanced through the introduction of automatic and interactive theorem proving. However, most of these techniques only work under restrictive assumptions that need to be discharged before a verification result can be transferred down to the machine level. For instance, verification techniques for concurrent systems usually postulate (a) fair scheduling, (b) the commutability of transitions in an interleaved sequence between adjacent synchronization points, and (c) compiler correctness for a high-level language with communication primitives.

In this paper, I present the concurrent model *Communicating User Programs* (CoUP) that describes processes running on top of the microkernel VAMOS. Moreover, I show that this model indeed simulates the formal specification of VAMOS. This simulation proof comprehends the three properties fairness, commutability, and compiler correctness, which are often just postulated. Thus, I bridge the semantic gap between the kernel specification and my abstract model CoUP.

The context of my work is the Verisoft project [1], which aims at a verified system stack—starting from hardware via a microkernel and a user-mode operating system up to applications. My work substantially relies on the following results of this project: (a) A small kernel has been developed. (b) Leinenbach and Petrova provided a proven correct compiler from a C variant into assembly language. (c) Dörrenbächer and I extended these sequential language semantics with suitable communication primitives for kernel interaction. (d) Dörrenbächer developed a formal kernel specification with concurrent processes operating on assembly language.

In Sect. 2, I briefly summarize these results. Sect. 3 is concerned with the CoUP model. In Sect. 4, I sketch the simulation theorem for CoUP. Sect. 5 deals with related work. Conclusions and further work are found in Sect. 6.

* 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.

2 Background

The Microkernel. VAMOS has been implemented on a RISC processor. The kernel manages a dynamically changing number of assembly processes and schedules them according to their priority. The priority-based round-robin scheduler can suspend and resume the processes after each machine instruction. VAMOS provides a separate bounded memory to each of its processes and acts as a mediator for all communication between the processes and with external devices.

The processes interact with the kernel via the application binary interface (ABI). The core of this interface is a specific assembly instruction called *trap* that causes an internal interrupt, which activates VAMOS. The trap instruction features an immediate constant as argument; additional arguments are conveyed to the kernel through the registers of the process. By *kernel call*, we refer to a trap number and the corresponding arguments from the process registers.

Most of the kernel calls are reserved for *privileged processes*. However, any process might use inter-process communication (IPC). 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.

A Correct Compiler for an Extensible Language. Leinenbach and Petrova [2] provide a proven correct compiler that translates programs of the perfectly type-safe C variant *C0* into the assembly language of the 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 arithmetics. In spite of that, *C0* still features dynamic memory allocation and inlined assembly.

For *C0* and the assembly language, two small-step transition semantics have been formalized in Isabelle/HOL. A *C0* program is represented by its static variables, types and function definitions. A configuration during execution is a pair of the remaining program $s_{C0}.prog$ and the current state of program variables $s_{C0}.mem$. On assembly level, there is no separation of instructions and data. A program counter points into a linear memory and interprets as an instruction whatever is found there. The configuration consists of the program counter, a file of general-purpose registers and a bounded amount of linear memory. The transition relations δ_{C0} and δ_{asm} of both semantics are partial functions.

Leinenbach has formally proven compiler correctness by stepwise simulation. A simulation relation *consis* 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} . This relation is parametrized over an *allocation function* mapping the current variables to memory locations. With these prerequisites in place, we state:

Theorem 1 (Compiler Correctness). *Assuming that s_{C0} represents a well-typed *C0* state, there is no run-time error in the next step, i. e., $\text{defined}(\delta_{C0}(s_{C0}))$, and there are sufficient resources, the following statement holds:*

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

A notable C0 feature is the inline-assembly statement. Though its semantics is unspecified, the compiler literally embeds the given code into the compilation. Using this hook, I have extended the sequential semantics by kernel communication primitives, which implement the kernel calls for C0. The primitives are functions with inlined assembly code using the ABI (see Sect. 3 for an example).

Modelling Processes. We have just learned of two kinds of processes: assembly processes using the ABI to interact with the kernel, and C0 processes interacting with the kernel via the kernel primitives. Based on the observation that the kernel primitives are implemented using the ABI, we can state that all processes interact with VAMOS only via this well-defined interface. We describe the ABI by the output alphabet Ω_{proc} specifying all kernel calls, and by the input alphabet Σ_{proc} comprising all possible kernel responses. Additionally, Σ_{proc} contains inputs demanding the process to alter its memory size. While the interface is generic, the internal process configurations $s \in \mathcal{S}$ are specific to the process model. Consequently, the output function ω that interprets the current configuration and the transition function δ are model specific. Dörrenbächer and I have defined the process semantics by the Moore automata $(\mathcal{S}_{\text{asm}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \omega_{\text{asm}}, \delta_{\text{asm}})$ for assembly processes and $(\mathcal{S}_{\text{C0}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \omega_{\text{C0}}, \delta_{\text{C0}})$ for C0 processes. In Sect. 3, I give a more detailed insight into the peculiarities of the C0-process model.

The Kernel Specification. The specification of VAMOS embeds a number of assembly processes. It is defined by the Moore automaton $(\mathcal{S}_v, s_v^0, \hat{\Sigma}, \hat{\Omega}, \omega_v, \delta_v)$ with the state space \mathcal{S}_v containing the initial state s_v^0 , the input alphabet $\hat{\Sigma}$ and the output alphabet $\hat{\Omega}$ for the device communication, the function ω_v that determines the output from the current state, and the transition function δ_v .

A configuration $s_v \in \mathcal{S}_v$ comprises (a) a partial function $s_v.\text{procs}$ that maps the process identifiers (PIDs) of the currently active processes to their assembly states, (b) scheduling data structures $s_v.\text{schedds}$, and (c) control data structures like a mapping from PIDs to priorities, a mapping from PIDs to access rights, and information for device communication.

Our kernel uses a memory-mapped I/O for device communication. The output alphabet $\hat{\Omega}$ comprises read and write accesses to device addresses, and input alphabet $\hat{\Sigma}$ consists of interrupt lines and optionally incoming data. Hillebrand *et al.* [3] have described our device interface in detail.

A kernel transition has up to three phases: (1) The current process is advanced if present. (2) A possible timer interrupt is handled. That means to increase the clock-tick counter and the consumed time of the current process as well as to wake up the waiting processes with now elapsed timeouts. (3) Occurred interrupts are delivered to their corresponding drivers if they are waiting or otherwise saved in the internal data structures for a later delivery.

This specification fully describes the semantics of the kernel. Dörrenbächer has developed this model and currently verifies that the implementation of VAMOS meets this specification. However, this model is not appropriate for the verification of user programs written in C0. For this purpose, I have developed the CoUP model, which I present in the next section.

3 Communicating User Programs

The key motivation for the concurrent model CoUP is a process model that supports the C0 program semantics. However, I intend to justify this process abstraction with respect to the assembly processes of the kernel specification. Here, I face the challenge that the process models have different granularity, i. e., a C0-process step usually consists of many assembly-process steps. Thus, a C0 process may be scheduled during the execution of a single statement. Hence, I first abstract from the VAMOS scheduler and then introduce the C0 processes.

Abstracting from the Scheduler. Similarly to the kernel specification, the CoUP model is a Moore automaton $(\mathcal{S}_{vc}, s_{vc}^0, \hat{\Sigma}, \hat{\Omega}, \omega_{vc}, \Delta_{vc}, \mathcal{R}_{vc})$, where \mathcal{S}_{vc} represents the state space containing the initial state s_{vc}^0 , $\hat{\Sigma}$ and $\hat{\Omega}$ are the input and output alphabets for device communication, ω_{vc} denotes the output function, Δ_{vc} is the transition relation, and \mathcal{R}_{vc} represents the *set of runs*.

The notable difference of this automaton is the non-determinism caused by the abandoned scheduler. Consequently, the transition relation is no longer functional. Moreover, we introduce the new notion of runs: While it is advantageous to abstract from the particular scheduler policy in VAMOS, we would like to preserve the more abstract property of fairness. It is impossible to encode this property in the transition relation of the model; we can only formulate it over an infinite sequence of transitions. I define a *run* of our model as an infinite transition sequence that satisfies the fairness property.

The state space of CoUP inherits its most components from the kernel specification. Only two components change: The process model becomes a parameter for CoUP, i. e., the component $s_{vc}.procs$ of a CoUP state s_{vc} is a partial function from PIDs to a generic state space \mathcal{S}_{proc} for processes, usually instantiated with \mathcal{S}_{up} , as defined below. Moreover, the scheduling data structures are replaced by a current-process indicator. The current process is retained 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, I fix this process beforehand instead of including transitions for all ready processes in the transition relation.

The transition relation is a set of triples comprising an input, the state before and the state after a transition. This relation resembles the three phases that we know already from the kernel specification.

Finally, I define the set of runs. A run is an infinite sequence of transitions that fulfils certain side conditions. I represent transition sequences as a tuple consisting of two functions *states* and *inputs* that map the step number to the state and the input, respectively. Naturally, a transition sequence starts with the initial state and the transition relation holds for adjacent states. Ultimately, I constrain the set of possible runs such that we preserve scheduler fairness, i. e.,

$$\mathcal{R}_{vc} = \{ (states, inputs) \mid states(0) = s_{vc}^0 \wedge \\ \forall i : (inputs(i), states(i), states(i+1)) \in \Delta_{vc} \wedge \\ isFair(inputs, states) \}$$

```

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. 1. Implementation of function `vc_process_clone` from the kernel library

I formulate fairness as liveness for the processes and do not quantify the progress of the processes with respect to time slices. Let us assume that an arbitrary, fixed process *pid* has the maximum priority infinitely often while the timer interrupt is active. A run 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.

Processes. Our user programs are implemented in C0. However, the pure C0 semantics cannot generate traps for the communication with the kernel. Hence, I have extended the original C0 semantics with a special kernel library. This library comprises functions that use inlined assembly code to implement the kernel calls. For example, Fig. 1 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.

The sequential C0 semantics implicitly assumes sufficient resources, which cannot be assumed for C0 processes. Hence, I have extended the state s_{C0} of the C0 semantics by the currently occupied memory size $s_{C0}.msize$ and defined the transition relation δ_{C0} such that it gets stuck for insufficient resources. We denote the set containing all states of C0 processes by \mathcal{S}_{C0} . Based on this state space, I have defined the output functions ω_{C0} and the transition relation δ_{C0} . For space restrictions, I cannot present the full definitions but I depict an exemplary excerpt of the formal definition for the call `process_clone` in Fig. 2.

For the C0 state s_{C0} , we consider the first statement of the program $s_{C0}.prog$. If that is a function call to `vc_process_clone`, the output of ω_{C0} is a pair of the constant `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 $s_{C0}.mem$ at the address where the left-value e is stored with the corresponding error code (-4) and removes the function call from the remaining program.

Note that not every assembly process can be represented as a C0 process. Although all our user programs will be implemented in C0, we would like to show safety properties such that an application will work correctly even in the

$$\begin{aligned}
s_{C0}.prog &= "e = vc_process_clone(e_0); r" \\
\implies \omega_{C0}(s_{C0}) &= (process_clone, get_val(s_{C0}.mem, e_0))
\end{aligned}$$

$$\begin{aligned}
s_{C0}.prog &= "e = vc_process_clone(e_0); r" \\
\implies \delta_{C0}(err_unprivileged, s_{C0}) &= s_{C0} \left[\begin{array}{l} mem := set_val(s_{C0}.mem, e, -4) \\ prog := r \end{array} \right]
\end{aligned}$$

Fig. 2. Formal definition of the output and the transition function of C0 processes for the call `process_clone`

presence of malicious assembly processes. Consequently, both process models must equally coexist in the CoUP model. I combine the state spaces of assembly processes and C0 processes by a disjoint union: $\mathcal{S}_{up} = \mathcal{S}_{asm} \uplus \mathcal{S}_{C0}$. The associated output and transition functions ω_{up} and δ_{up} simply work as mediators, selecting the corresponding function depending on the kind of state.

4 Simulation Theorem

In this section, I show that the CoUP model simulates the kernel specification. Recall that I abandoned the scheduler. Hence, I prove that the transition sequences of the resulting, non-deterministic model still contain the transitions of the specification. Moreover, I show that we can shift scheduling decisions to statement boundaries in this abstract model. Then, we can consider assembly code sequentially, compile C0 programs into assembly code, and link them against the kernel library. In the remainder of this section, I detail the three steps.

4.1 Justifying the Scheduler Abstraction

Here, I assume a CoUP model with solely assembly processes and prove that the transitions of the kernel specification are simulated by CoUP. For this proof, I relate the state spaces of both models. This relation is functional and straightforward because only the scheduling data structures are abandoned. We denote this abstraction function as $abs \in \mathcal{S}_v \rightarrow \mathcal{S}_{vc}$. Now I show the simulation in two steps: At first, I show that every transition sequence of the kernel specification has an equivalent in CoUP. At second, I proof that fairness holds.

Theorem 2 (Equivalence of Transition Sequences). *The initial states are equivalent, i. e., $abs(s_v^0) = s_{vc}^0$, and their equivalence is preserved after each transition of the specification, i. e.,*

$$(inputs, abs(s), abs(\delta_v(inputs, s))) \in \Delta_{vc}$$

Proof Insights. I proved this statement mainly by repeated case distinctions. As we could just glance at the transition relations in this paper, I can only summarize a few insights from my proof. All in all, the verification was straightforward

because I reused many definitions from the specification in CoUP. The biggest difficulty I faced was in the verification of IPC because of a considerably simpler modelling in CoUP, which became possible after the scheduler was abstracted. \square

Theorem 3 (Fairness). *The VAMOS scheduler is fair, i. e., with the set \mathcal{R}_v containing the infinite transition sequences of the specification, we can state*

$$\forall (states, inputs) \in \mathcal{R}_v : isFair(inputs, abs \circ states)$$

Proof Sketch. An arbitrary, fixed process pid can either be inactive, waiting, or ready. In the first case, Condition (a) of the fairness predicate 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 time-slices are bound by a fixed value. Hence, the process pid will eventually 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, the VAMOS 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 I have shown fairness for all processes in this queue. \square

I just sketched the simulation proofs here on a very abstract level. However, I have formally developed both proofs in Isabelle/HOL within ten person months.

4.2 Shifting Scheduling Decisions

Let us assume that there is a currently computing assembly process pid , which represents a compiled C0 program. When we shift scheduling decisions to statement boundaries, we reorder the interleaved transitions of the global system such that we obtain a sequential trace of assembly steps for this process. This reordering is confluent iff the transitions do not interfere with each other.

Theorem 4 (Reordering). *For a compiled C0 program, we can shift scheduling decisions to statement boundaries.*

Proof Plan. Trace theory can be employed for this proof. In short, transitions can be commuted if they are independent of each other. Such transitions either change the system but not the considered process pid or exclusively this process. Non-commutable transitions, called *synchronization points*, affect simultaneously the kernel data structures and the considered process pid . The most prominent representative is the execution of a trap instruction of process pid : The current process is pid , its output function ω_{asm} returns an output ω different from ε , and the kernel generates some input σ for the process.

Consequently, we can merge assembly instructions into one transition as long as they contain up to one synchronization point. Ordinary C0 statements are translated into exception-free assembly steps and the kernel library does not interact with the kernel more than once per library function.

Unfortunately, we have so far neglected the kernel’s ability to 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 C0 processes 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 regard this process as a C0 process 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 regard all other processes as C0 processes. \square

4.3 Compiling and Linking

So far, the argumentation has been quite generic and can be reused for other language semantics. The last step is C0-specific: We examine compilation and linking. The user programs are implemented in pure C0 without inlined assembly code but may contain function calls to the kernel library. For simplicity, programs are linked 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 inlined assembly code on the other hand.

Theorem 5 (Process Simulation). *C0 processes simulate assembly processes.*

Proof Plan. 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. 3 depicts our course of action when a library function is called in some C0 state s_{C0}^n . We relate this C0 state to an assembly state s_{asm}^b using the simulation relation *consis* of compiler correctness.

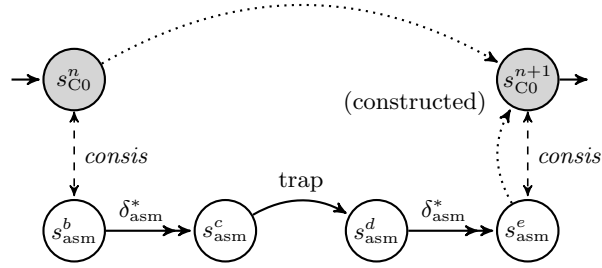


Fig. 3. Verification scheme for inlined assembly portions in the kernel library

Now, we execute the inlined assembly code starting in s_{asm}^b . When the code reaches the trap instruction in state s_{asm}^c , the assembly process has to signal the same output to the kernel as the C0 process does in s_{C0}^n . Finally, we arrive at the end of the inlined assembly portion in a state s_{asm}^e . At this point, we inspect the changes of the inlined assembly code between s_{asm}^b and s_{asm}^e . If the memory differs only for visible variables in a type-safe manner, we can construct a corresponding C0 state s_{C0}^{n+1} from s_{asm}^e and s_{C0}^n . Examining this new C0 state, the result value from the kernel should indeed be stored as it is specified in the C0-process model. \square

A similar problem has successfully been proven by Starostin and Tsyban [4].

5 Related Work

Bevier [5] reports on an early successful attempt to verify an operating-system kernel. However, this kernel is very limited compared to modern microkernels.

Many recent projects report on verification efforts on modern microkernels. Among them are VFiasco [6] and its successor Robin [7], L4.verified [8], EROS [9] and its successor Coyotos [10], as well as the FLINT project [11]. Though these projects may have achieved great results, they all focus on microkernel verification but do not ascend towards the verification of user programs.

Ábrahám *et al.* [12] developed a proof system for a multithreaded Java variant. This system provides a more fine-grained model of concurrency than ours but the language is more academic while our approach is application driven.

Recently, Hobor *et al.* [13] have proposed a verification method for concurrent programming languages. The group assumes a fairly abstract language with threads that communicate using locks while competing for a shared memory. This model is excellent for a multithreaded program but does not suit our environment with synchronous communication through a microkernel. Besides the differences of the communication model, this work postulates fair scheduling and correctness of the locking primitives.

6 Conclusion and Future Work

The theory of concurrent systems is long established and well-studied. However, most verification techniques for concurrent systems rely on unproven assumptions about the underlying system. In this paper, I presented a concurrent model for user processes running on top of a particular microkernel and proved that all assumptions for this model indeed hold for the kernel specification. While my solution features a less general concurrency model in comparison to traditional methods, my approach is tailor-made for a concrete, realistic problem.

The presented proofs are partially formalized in Isabelle/HOL. I intend to complete this effort by formalizing my proof plans for Theorems 4 and 5. Additionally, I plan the verification of an operating-system component using my model.

Acknowledgements. I thank Burkhart Wolff, Sarah Hoffmann, and Norbert Schirmer for inspiring discussions, as well as Irena Dotcheva for correcting my English.

References

1. Hillebrand, M.A., Paul, W.J.: On the architecture of system verification environments. In: Haifa Verification Conference, Springer (2007) 153–168
2. Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Systems Software Verification, Springer (2008)
3. 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
4. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Systems Software Verification, Springer (2008)
5. Bevier, W.R.: Kit and the short stack. *J. Autom. Reasoning* **5**(4) (1989) 519–530
6. 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
7. 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
8. 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**(4) (2007) 3–11
9. Shapiro, J.S., Weber, S.: Verifying the EROS confinement mechanism. In: IEEE Symposium on Security and Privacy. (2000) 166–176
10. 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
11. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: TPHOLs, Springer (2007) 189–206
12. Ábrahám, E., de Boer, F.S., de Roever, W.P., Steffen, M.: An assertion-based proof system for multithreaded Java. *Theor. Comput. Sci.* **331**(2-3) (2005) 251–290
13. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: ESOP. Volume 4960 of LNCS., Springer (2008) 353–367