

Preemption Abstraction

A Lightweight Approach to Modelling Concurrency

Erik Schierboom³, Alejandro Tamalet^{1*}, Hendrik Tews^{1**},
Marko van Eekelen¹², and Sjaak Smetsers³

¹ Digital Security Group, Radboud Universiteit Nijmegen

² Faculty of Computer Science, Open University

³ BliXem Internet Services

{eschierb,tamalet,tews,marko,s.smetsers}@cs.ru.nl

Abstract. This paper presents the *preemption abstraction*, an abstraction technique for lightweight verification of one sequential component of a concurrent system. Thereby, different components of the system are permitted to interfere with each other. The preemption abstraction yields a sequential abstract system that can easily be described in the higher-order logic of a theorem prover. One can therefore avoid the cumbersome and costly reasoning about all possible interleavings of state changes of each system component. The preemption abstraction is best suited for components that use preemption points, that is, where the concurrently running environment can only interfere at a limited number of points.

The preemption abstraction has been used to model the IPC subsystem of the Fiasco microkernel. We proved two practically relevant properties of the model. On the attempt to prove a third property, namely that the assertions in the code are always valid, we discovered a bug that could potentially crash the whole system.

1 Introduction

In this paper we focus on the verification of the following kind of systems: a component \mathcal{C} is running in a concurrent environment \mathcal{E} , where \mathcal{E} interferes asynchronously with the component \mathcal{C} by, for instance, changing some state variables of \mathcal{C} . The goal is to prove some specified property about the component, regardless of how the environment behaves.

This kind of problem appears for instance in operating-system verification. Every recent operating system permits several threads of execution running in quasi parallel, even on a system with only one processor core. Typically each such thread might invoke any operating-system call. Nevertheless, the effects the different threads might have on each other are relatively limited. For the verification of the operating system, it is therefore often sufficient to consider only

* Sponsored by the Netherlands Organization for Scientific Research grant 612.063.511.

** Supported by the European Union through PASR grant 104600.

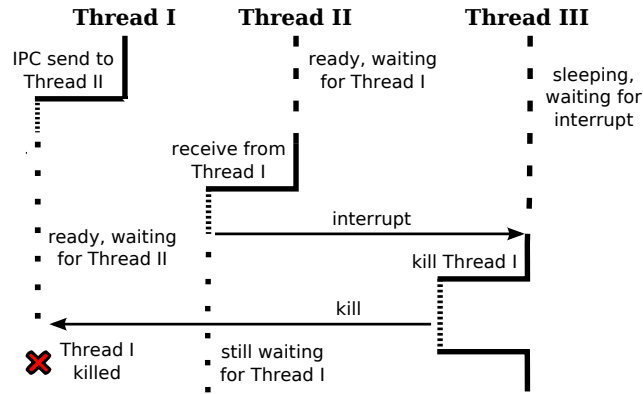


Fig. 1. Environment (thread II + III) asynchronously interfering with thread I. The zigzags in the lines represent system calls: the threads are executing user code in the solid lines and operating-system code in the dotted lines. Dashed lines and separated dots indicate that the thread is not scheduled.

one thread of execution, and to model all the threads that can asynchronously affect the given thread as some kind of environment.

As an example, Figure 1 shows three threads. Initially, thread I and thread II want to exchange a message via inter-process communication (IPC), while thread III is sleeping. Thread II and thread III can be considered as the environment of thread I, that is, they can asynchronously affect thread I. When thread I performs a system call in order to send a message to thread II, the environment could react in several ways (where only the last one is displayed in Figure 1):

- The environment could do nothing, corresponding to a situation where thread II never performs the system call necessary to receive from thread I.
- The environment could engage in IPC with thread I, corresponding to a situation where thread II successfully receives the message from thread I.
- The environment kills thread I, as displayed in Figure 1. Here thread II starts the system call to receive from thread I, but then an external interrupt wakes up thread III. Thread III immediately gets scheduled (for instance because it has a higher priority) and kills thread I.

It is important to notice here that the number of different effects that the environment can have on thread I, is rather limited. Although every thread runs arbitrary user code, there is only a limited number of system calls and only few of them can have an effect on thread I.

Only few operating-system kernels are fully interruptible, meaning that re-scheduling of a different thread can occur at every point in every kernel procedure. Maintaining consistency of kernel data structures for a fully interruptible kernel is difficult, therefore many kernels disable rescheduling or even interrupts over large portions of the kernel. When real-time properties are a concern, a kernel design with *preemption points* is sometimes used. In this design, interrupts (and therefore rescheduling) are generally disabled, except at well-defined

points —the preemption points. Pending interrupts are then delivered only at these points. Kernel data structures are synchronized before any preemption point so that rescheduling a different thread (which might engage in different kernel activities) can be done without danger of corruption.

In this article we describe and use the *preemption abstraction*, an abstraction technique tailored for this kind of systems. The technique has been developed for creating and verifying models in the higher-order logic of an interactive theorem prover. The preemption abstraction is equally well applicable in a model-checking environment, although its benefits there will not be as remarkable as in interactive theorem proving. We used the abstraction technique in the modeling and verification of the inter-process communication (IPC) facilities of the microkernel Fiasco [Hoh01,Hoh98,HP01]. Our verification attempt identified one programming error, although the part of the IPC subsystem that was modelled was thoroughly tested and in daily use. The bug could only be triggered when a specific interrupt occurred precisely in a very short time frame during the execution of the IPC system call. It was therefore so unlikely to trigger the bug that it could have stayed unidentified for decades.

This paper is organized as follows. The next section describes the preemption abstraction while Section 3 describes Fiasco, in particular its IPC subsystem. In Section 4 the PVS model is discussed, with emphasis on the application of the preemption abstraction. Section 5 comments on the properties that were verified and on the programming error that was found. In Section 6 we evaluate the case study and give pointers to future work. Finally, Section 8 draws conclusions.

2 The Preemption Abstraction

Consider a parallel system \mathcal{S} , as exemplified in the introduction, with the following properties. \mathcal{S} consists of an arbitrary number of threads and each thread consists of a sequence of atomic blocks. Between each two atomic blocks there is a preemption point, in which no computations and state changes are performed (in practice a preemption point consist of one or two NOP instructions). For each atomic block, each thread acquires a global lock, which is released during the preemption points. Thus, a computation of the whole system consists of one sequential interleaving of all the atomic blocks. Apart from the sequential interleaving, the threads may interfere in arbitrary ways, for instance, a thread t_1 may change the state of another thread t_2 . Because of the sequential interleaving, t_2 is of course waiting in a preemption point when t_1 changes its state.

The preemption abstraction focuses on one selected thread t . All other threads are considered as the environment of t . When t is waiting for the global lock in a preemption point, any thread from the environment can change the state of t . All such potential changes are collected in the set of side effects SE . For real systems this set would typically have a small finite cardinality, but the correctness of the abstraction does not depend on that. In this work we assume that the events in SE are independent, however, the abstraction could still be applied if such dependencies are made explicit on the model.

In the following we consider (finite) lists of side effects taken from SE. Note that one particular side effect can occur multiple times in such a list.

The preemption abstraction \mathcal{A} of the system \mathcal{S} consists *only* of the thread t with the following changes:

- The *preemption-point function* is substituted for all preemption points in t . The preemption-point function nondeterministically chooses an arbitrary list of side effects and executes it.
- The global lock, its acquisition and release are abstracted away.

In the preemption abstraction all the other threads of \mathcal{S} that form the environment of t are condensed into the preemption-point function.

The preemption abstraction \mathcal{A} is a sequential model of \mathcal{S} that faithfully models the behavior of the thread t . Under the assumption that there are no dependencies between the threads the abstraction suffices to prove arbitrary (functional) properties of t that can be proved in \mathcal{S} . Since it takes the point of view of a single thread, the abstraction cannot be used to prove properties about cooperating threads. The abstraction is sound in the sense that every property proved for t in \mathcal{A} also holds in \mathcal{S} . The soundness crucially depends on the completeness of the set of side effects SE.

The main advantage of the preemption abstraction \mathcal{A} is that it is a sequential model, consisting of only one thread. For a description of its behavior one does not have to consider different interleavings of atomic blocks. The abstraction \mathcal{A} can therefore be conveniently described as a functional model in the higher-order logic of an interactive theorem prover. In contrast, modelling the behavior of \mathcal{S} with all possible interleavings of its threads in higher-order logic would be a major hassle. The preemption abstraction is therefore absolutely necessary in order to verify nontrivial systems \mathcal{S} in an interactive theorem prover.

The preemption abstraction can also be applied in a model-checking context. Because model checkers have built-in support for parallel systems the sequentiality of the preemption abstraction is not an advantage per se. However, the reduction of the system \mathcal{S} with its arbitrarily many threads to just one thread should make the state space much smaller. Using the preemption abstraction for model checking remains future work.

3 Interprocess Communication in Fiasco

The Fiasco microkernel belongs to the L4 microkernel family. It has been developed since 1998 at TU Dresden, Germany. It is mainly written in C++ with some inline assembly and assembly short-cuts for the most performance critical system calls. In a microkernel based system many operating-system services are implemented as separate modules, which are running as normal application programs. Therefore inter-process communication (IPC) is often the bottleneck of microkernel based systems. With very stringent optimizations, the L4 microkernel interface and some of its implementations remedied this problem, achieving

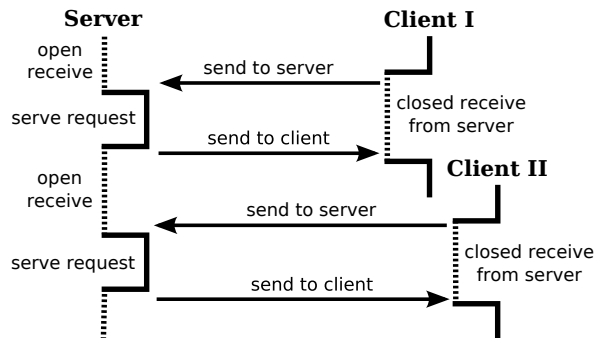


Fig. 2. Typical communication pattern for applications running on an L4 microkernel. As before solid lines indicate user code and dotted lines indicate operating-system code.

performance within 5% of traditionally designed systems [HHW98]. The L4 family (and other microkernels) is therefore sometimes referred to as a microkernel of the second generation.

The Fiasco microkernel implements processes, threads, address spaces, inter-process communication and delegation of memory resources. The only device that the kernel controls itself is the interrupt controller. Drivers for all other devices, such as hard disks, graphic cards and keyboards run outside of the kernel as normal application programs.

IPC will play an important role for this paper, so let us elaborate a little bit on it. IPC in the L4 interface is optimized for the common case of client-server communication. There is just one system call for IPC, whose precise behavior can be modified via certain parameters. IPC in Fiasco is always synchronous, that is, sender and receiver must perform a rendezvous. If either the sender or the receiver is not ready, the other party blocks. In general the IPC system call always performs a send operation followed by a receive. Both the send and the receive operation are optional and can be disabled via parameters to obtain a send-only or receive-only IPC system call. If the send operation is enabled it always sends to a specified destination thread. The receive operation can be either *open* or *closed*. In an open receive any IPC partner is accepted, while in a closed receive only messages from one specified thread are accepted. Both the send and the receive operation always transfer two registers plus, optionally, some memory contents. If some memory is copied it is called *long IPC*, otherwise *short IPC*. Typically short IPC prevails and shared memory is used for bulk data transfer. The time the IPC system call blocks in either the send or the receive operation can be controlled via timeout parameters. As special cases the timeout can be zero (abort IPC if the partner is not ready) or infinite (no timeout).

Figure 2 shows how the IPC operation is exploited in client-server communication. At the beginning the server blocks with infinite timeout in an open receive until client I starts a complete IPC call. This call consists of a send operation and a closed receive, both with the server as IPC partner. When the send operation from client I to the server is complete, the server finishes its IPC

system call and starts working on the client request. Meanwhile, client I blocks in a closed receive (typically with infinite timeout) until the server answers.

When the server finishes working on the request, it starts a new complete IPC system call. In the send operation it sends its answer back to client I. Client I thereby finishes its IPC system call and continues normal computation. After sending the answer, the server blocks in an open receive waiting for the next client. The server can thus be programmed in a loop with one IPC system call as last statement of the loop. At server boot time, just before entering its main loop, the server does an open receive without send operation.

In Fiasco IPC is implemented such that the sender is the active part. That is, the sending IPC partner performs the necessary locking and copies the message. The receiving IPC partner simply waits until some sender finished its job.⁴

In Fiasco, thread ID's are 64-bit numbers. They are used to denote potential senders and receivers. There are two special thread ID's: the invalid thread ID, sometimes referred to as null-thread ID, and the nil-thread ID. The nil-thread ID can for instance be used in a closed receive with some timeout. As effect the thread will sleep until the timeout elapses.

4 The Model

This section describes our model of Fiasco's inter-process communication with special emphasis on the abstraction described in Section 2.

For the formalization we chose the theorem proving approach and, in particular, we used the PVS theorem prover [OSRS01]. Section 7 describes other works that used the model checking approach to model the same subsystem.

PVS consists of a specification language based on higher-order logic with dependent types and predicate subtyping, and tools to create and manipulate proofs. Its intuitive syntax is reminiscent of functional languages like Haskell.

The code that had to be modelled was written in a small subset of C++: mainly assignments, conditionals and method calls. We reduced it even more by abstracting most loops and splitting functions with side effects into a state transformer plus a pure function that returns a value. This resulted in a shallow embedding of the C++ sources in PVS.

4.1 Key Abstractions

In a real system executing on the Fiasco microkernel, many threads can run in parallel, and each one can start an IPC system call. Therefore, the IPC code in the kernel potentially runs in parallel with itself many times. In order to obtain a sequential model that can be easily described in PVS, we applied the preemption abstraction as explained in Section 2.

⁴ An exception are interrupts that are mapped into an IPC message to the thread that registered for that interrupt. In this case the receiving thread is active. However, interrupt IPC is not considered throughout this paper.

As the first step we identified the set of side effects **SE**. When a thread t performs an IPC operation other threads can modify the state of t in the following way: (1) the thread t can be killed, (2) a timeout can occur meaning that t should not wait any longer for an IPC partner to become ready, (3) the IPC operation of t can be canceled or (4) a receiver can become ready, meaning that t can proceed with the send part of the IPC. The side effects are modeled in PVS with the type **PreemptionAction** and the function **doPreemptionAction**, as we will explain in Section 4.2 below.

As a second step we focused on the IPC code of just one thread, ignoring the rest of the system. The preemption points are replaced by the preemption-point function, which is formalized in PVS by **preemptionPoint**, see Section 4.2 below. Note, that after applying the preemption abstraction there is no scheduler left in the model. The only effect the scheduler could have is that our thread t remains for a longer time in some preemption point and that therefore some more side effects accumulate.

Independently from the preemption abstraction, we decided to focus on the core functionality of IPC. We only model short IPC between real threads (no preemption IPC, no interrupt IPC, no long IPC). Note that we do model timeouts in an abstract way without any notion of time in the model: A timeout side effect can occur in any preemption point.

4.2 PVS Specification

In PVS a theory is a module that encapsulates definitions and properties. It provides a means to hierarchically decompose a specification. Our work is composed of several theories with simple dependencies among them. The theories **state** and **ipc** contain the model and will be discussed in this section. For reasons of clarity and space, we will restrict ourselves to some relevant, slightly simplified extracts of our model. The complete specification can be obtained via <http://www.cs.ru.nl/~tamalet/>.

We define **ThreadPointer** as an uninterpreted type, which essentially represents an arbitrary set. This set should have at least two elements, which will be enforced by an axiom. We say that **null** is a **ThreadPointer**, and declare **NonnullTP** as the set of non-null thread pointers. The **nil_thread_ptr** constant points to the special nil thread (see Section 3), used to encode send-only or receive-only IPCs

```

ThreadPointer: TYPE
not_empty_or_single: AXIOM  $\exists$  (tp1, tp2: ThreadPointer): tp1  $\neq$  tp2
null: ThreadPointer
NonnullTP: TYPE = { tp: ThreadPointer | tp  $\neq$  null }
nil_thread_ptr: NonnullTP

```

Fiasco stores the status of a thread in a bit vector. We have represented the flags of this vector that are relevant to our model as a record with boolean fields. A complete description of the status flags can be found in [Hoh02].

```

ThreadStatus: TYPE = [# ready, cancel, dead, busy, invalid,
  polling, receiving, ipc_in_progress,
  send_in_progress, transfer_in_progress: bool #]

```

Each thread is composed of a `status`, a `partner` to engage with in IPC and a list of senders (named `senders_waiting`) containing the senders that are queued if the receiver is busy. As explained in Section 3, the sender is the active part in an IPC, and one of the actions a sender performs is locking the receiver. In our abstract model, it is sufficient to know which sender owns the lock. This results in the following representation of threads.

```

Thread: TYPE = [# status: ThreadStatus,
  partner, lock: ThreadPointer, senders_waiting: list[NonNullTP] #]

```

Though the status flags can be set/cleared individually, one usually considers a certain combination of flags to check whether a thread is in a specific state. For instance, to determine whether two threads are engaged in IPC, the following tests are necessary:

```

inIpc(snd, rcv: NonNullTP)(s : System): bool =
  LET rcv = s'threads(rcv), rcv_stat = rcv'status IN
    rcv_stat'transfer_in_progress ^ rcv_stat'ipc_in_progress ^
    ¬rcv_stat'cancel ^ rcv'partner = snd

```

PVS-functions are explicitly parameterized with a `System` object representing the global state of the machine. Moreover, each function will produce the modified global state as a result. This state is defined as follows:

```

System: TYPE = [# current: NonNullTP, threads: [NonNullTP → Thread],
  error, timeout, fail: bool, seed: nat #]

```

The field `current` is a pointer to the active thread, `threads` is a 'dereference' function yielding the threads of the system, `error` and `timeout` indicate if an error or a timeout occurred, respectively, and `fail` is set if one of the assertions failed. The field `seed` is explained below.

The manipulation of state information makes specifications needlessly complex. However, with a suitable set of helper functions, one can easily avoid an explicit state object. Particularly, the following composition operation appears to be convenient in our description.

```

SystemFun: TYPE = [System → System]

```

```

>>(s1, s2: SystemFun): SystemFun = λ (s: System):
  LET s1s = s1(s) IN IF s1s'error THEN s1s ELSE s2(s1s) ENDIF

```

This operation resembles standard function composition. Observe that the second function will not be applied if the first one resulted in an error.

In the first step of our approach the set `SE` of possible side effects is identified. This was done by careful examination of the possible effects concurrent threads can have on a each other, resulting in the following set of preemption actions:

```

PreemptionAction : TYPE =
{ kill,                % The partner is killed
  timeout,            % A timeout occurs
  ipc_cancelled,     % IPC has been canceled
  receiver_ready }   % The receiver becomes ready

```

Next, all preemption points are replaced by non-deterministically chosen list of preemption actions that are executed. Since PVS does not directly support non-determinism, we introduce the following auxiliary function:

```

generatePAs(n: nat): list[PreemptionAction]

```

This function is not further specified. In a proof, this means that we cannot assume anything about the actions appearing in the result list, hence it has to be considered as arbitrary. The argument `n` is necessary for technical reasons: by using different argument values each time `generatePAs` is called, different result lists will be produced. For, had we omitted this argument, `generatePAs` would have been treated as a constant, yielding the same unspecified list of preemption actions everywhere it is called. This explains the existence of the `seed` field in the system state. At each preemption point, the seed is passed to `generatePAs`, and it is incremented.

The effect of preemption actions on the system state is specified by the function `doPreemptionAction`:

```

doPreemptionAction(partner: NonNullTP, allow_timeout: bool)
  (act: PreemptionAction, s: System): System =
CASES act OF
  ipc_cancelled: sysThreadExRegs(s'current)(s),
  timeout:      IF allow_timeout THEN timeOut(s'current)(s)
                ELSE s ENDIF,
  kill:        kill(partner)(s),
  receiver_ready: IF s'current = partner THEN s
                  ELSE receiverReady(s'current, partner)(s) ENDIF
ENDCASES

```

The functions `sysThreadExRegs` and `kill` basically set the `cancel` and `dead` flags of the thread status vector, respectively, while `timeOut` sets the `timeout` flag of the system state. The function `receiverReady` sets the `ready` and `transfer_in_progress` bits on the sender and unsets `ready` on the receiver. Ensuring that the sender and the receiver are not the same whenever `receiverReady` is called was necessary to prove certain properties; see Section 5. In Fiasco, this is implicit since it doesn't make sense for a thread to engage in IPC with itself⁵.

Finally, we define `preemptionPoint` as the preemption-point function that executes a list of preemption actions.

```

preemptionPoint(partner: NonNullTP, allow_timeout: bool)
  (s: System): System =

```

⁵ And if it tries to, it will get deadlocked waiting for itself to become ready.

```

doPAs(partner, allow_timeout)(generatePAs(s' seed))(newSeed(s))

newSeed(s: System): System = s WITH [seed := s' seed + 1]

doPAs(partner: NonNullTP, allow_timeout: bool)
(pas: list[PreemptionAction])(s: System): System =
  reduce(s, doPreemptionAction(partner, allow_timeout))(pas)

```

The function `reduce` is a predefined list function, similar to `fold` or `fold_left` in other languages. In essence, `doPAs` composes the effects of the preemption actions occurring in the list.

These and other basic definitions form the `state` theory. The `ipc` theory contains the model of the C++ functions that implement Fiasco's IPC mechanism. The main function of this theory is

```

doIpc(rcv, snd: NonNullTP, has_rcv, has_snd: bool)(s: System): System =
  IF has_snd ^ has_rcv
  THEN doIpcSend (rcv, TRUE) >> doIpcReceive(snd)(s)
  ELSIF has_snd THEN doIpcSend(rcv, FALSE)(s)
  ELSIF has_rcv THEN doIpcReceive(snd)(s)
  ELSE s ENDIF

```

A few details of `doIpcSend` will be discussed later; the definition of `doIpcReceive` is unimportant for this paper.

5 Validating some Properties

This section is based on the PVS theories `prop_wakeup`, `prop_locks`, and `prop_assertions` containing our properties of interest.

Property 1: Receiver woken Consider the send part of an IPC call of a thread t_s that transfers data to a partner thread t_r . In Fiasco the sender is the active part, that is, t_r is sleeping during its receive operation. Sleeping here means that the `ready` flag of t_r is false, causing the scheduler to never select t_r . It is therefore essential that, after the send has been finished, the thread t_s wakes up its partner t_r , such that t_r can be scheduled again. This property is formalized as follows:

```

receiver_woken: LEMMA
  ∀ (partner: Non_Null_TP)(s: System):
    LET sSend = doIpcSend(partner)(s) IN
      ¬sSend'error ^ inIpc(sSend'current, partner)(sSend)
      ⇒ sSend'threads(partner)'state'ready

```

The property states that if after the execution of `doIpcSend` there is no error on the system state and the sender and the receiver are still engaged, then the `ready` bit of the receiver is set. The proof posed no difficulty and it consisted mainly of definition unfoldings and case distinctions.

Property 2: Lock removed Consider again a thread t_s that wants to engage in a send operation with t_r as receiver. Before actually starting the send, t_s obtains the lock of t_r to make sure that it is the only thread sending to t_r . After the send the lock must of course be released again.

```
lock_removed: LEMMA
  ∀ (rcv_ptr: NonNullTP)(s: System):
    ¬s'error ∧ ¬s'threads(rcv_ptr)'status'invalid ∧
    rcv_ptr ≠ nil_thread_ptr ⇒
      LET new_state = doIpcSend(rcv_ptr)(s) IN
        new_state'threads(rcv_ptr)'lock = null
```

The property has three requirements, namely, the state of the receiver must be valid, the receiver must not be the nil thread and there should be no error flagged on the initial system state. Under these conditions we were able to prove that after the execution of `doIpcSend`, the lock on the receiver is free.

To reduce the complexity of the proof, five lemmas were created. They assert that the lock is released on each of the possible path taken by `do_ipc_send`. This decomposition was also very helpful in making the proof more resistant to changes in the model.

Property 3: Assertions passed The Fiasco sources contain some assertions. When an assertion in the kernel is violated, the system simply halts. We included all the assertions that were expressible in our model, but some referred to things we had abstracted from, like the CPU lock, and thus were omitted. In total nine assertions were checked and it was in one them where the bug was found.

To find out if any of them could fail during a call to `sysIpc`, we added the field `fail` to the system state and we defined:

```
assert(b: bool)(s: System): System =
  IF b THEN s ELSE s WITH [fail := TRUE] ENDIF
```

Then the property was stated as shown next.

```
assertions_passed: LEMMA
  ∀ (rcv, snd: NonNullTP, has_rcv, has_snd: bool, s: System):
    ¬doIpc(rcv, snd, has_rcv, has_snd)(s)'fail
```

The function `doIpcSendPart` contained the assertion causing the failure.

```
doIpcSendPart(partner: NonNullTP, b: bool): SystemFun =
  tryHandshakeReceiver(partner) >>
  λ(s: System): assert(¬s'threads(s'current)'status'polling) >>
  [...]
```

The problem found is related to the `polling` bit, which is set on the sender when it has to wait for the receiver to become ready. Essentially, the sender *polls* the receiver at intervals to see if it has become ready. Once the receiver is ready and the handshake finishes successfully, this bit should be cleared.

When trying to prove that after a (successful) call to `tryHandshakeReceiver`, the `polling` bit is cleared, we found an execution path in the `doSendWait` function

(invoked by `tryHandshakeReceiver`) that did not clear it. After careful examination of the model, the author of that code was contacted and it was confirmed that indeed we had found an error.

But this was not the only complication we faced. There was also an assertion that could not be completely verified within our model due to the abstractions made on the sender. Since we did not model the sender as a separate thread, we could not prove that `inIpc` is commutative, that is, if the sender is engaged in IPC with the receiver, then the receiver is engaged with the sender. An axiom was added to overcome this problem.

Proving this property was quite laborious; 78 other lemmas were used directly or indirectly. The proofs were not intrinsically hard but cumbersome. The unfolding of some definitions resulted in proof sequents spanning hundreds of lines in the PVS prover. The following simple pattern can be identified in many of the proofs: unfold definitions and give names to intermediate states (to reduce the size of a sequent) as needed, then prove each branch using other lemmas if needed. Thanks to our lightweight approach to model concurrency, the number of branches was amenable to interactive theorem proving. The only proofs that needed induction were the ones concerning the list of actions that occur at a preemption point and the proofs dealing with the list of senders in the receiver.

6 Case Study Evaluation

In this section we share some reflections and lessons learned from our case study. We also comment on possible directions for future work.

Main lessons learned The case study has validated the applicability of the preemption abstraction approach as a lightweight formal proof method for concurrent code.

Using the proof assistant PVS, we modeled `sys_ipc`: the function that handles all inter-process communication focusing on the interaction between senders and receivers. While constructing the model we followed the source code (its structure and names) as much as possible. We focused both on a few key properties and on the assertions that were contained in the code. Furthermore, we abstracted from some important parts of the system, such as scheduling and Long IPC. Therefore, this case study cannot give a full formal proof of the studied system. However, the proofs of the studied properties significantly increased the confidence in the studied code and, when we found the bug, we could easily point out the corresponding place in the source code where the error occurred.

The code that was analyzed is about 3000 lines. The PVS model is about 2000 lines and the proof scripts are another 5000 lines long. Developing the proofs took 2 man-months but checking the proofs takes just a few minutes.

We want to emphasize the fact that the bug was found thanks to an assertion in the code. One usually thinks of assertions as just a runtime check mechanism, but they are more than that: they describe the intended behavior of the code. We used them to generate properties of our model of the system. Had the code not been instrumented with assertions, we would have probably missed the bug.

The soundness of our approach to model concurrency depends of course on the completeness of the list of actions that may occur at preemption points. We determined the possible events that the environment could have on thread at a preemption point by studying the source code. We are fairly confident that our list is exhaustive, however, a fully formal proof would also verify this assertion.

Applicability to systems without explicit preemption points The applicability of the preemption abstraction does not depend on the presence of explicit atomic blocks and preemption points in the software. On conventional hardware memory access is atomic, even in systems with multiple processors. For the preemption abstraction it is therefore not relevant whether there are possibly several threads running truly in parallel on several CPU's or not. The important point is, that at the level of memory access, all activity in the system is sequentialized. Therefore, one can think of a memory access as an atomic block with preemption points between memory accesses.

Under this interpretation, the number of preemption points will truly be tremendous. One clearly has to formulate the abstract model without writing out every invocation of the preemption-point function. This can easily be achieved with a higher-order combinator that inserts the preemption-point function after each memory access. A legitimate question is, whether it is still possible to verify any property in such a model. In general, the situation is admittedly hopeless. However, systems that have been designed to run in a truly parallel environment without the use of locks are far from the general case.

As an example let us consider a predecessor version of Fiasco that was fully preemptable. There, a timer interrupt could occur after each assembly instruction and induce the scheduling of a different thread. This new thread could potentially modify the state of the interrupted thread. This predecessor version of Fiasco was written in the lock-free programming style [Hoh01]: To modify a kernel data structure, a thread would first make a private copy, modify this private copy and finally write back the new version in an atomic way (for instance by using the compare-and-swap instruction). If the original data structure has been modified in between, it tries the same procedure again. This way, large portions of the code cannot be affected by parallel running threads, because it only operates on data structures that the other threads do not modify. The calls to the preemption-point function in the abstract model of such code can therefore be treated automatically in the verification environment.

Future Work A logical next step could be to extend the model and prove more properties. We would start by adding preemption and interrupt senders as well as long IPCs. It would also be interesting to prove the completeness of the set of preemption actions. This could be done by modelling all system calls and showing that any effect these calls can have on a running thread has already been considered. During the first phase of this work, we would have benefited from having a tool that, once configured, semi-automatically produces an abstract model. How to create a general tool that yields different models depending on the user's needs, is an interesting research topic with much potential.

7 Related Work

This work is based on the master’s thesis of Erik Schierboom [Sch07], in which a first version of the model was developed and the error was spotted.

Fiasco, and in particular its IPC subsystem, has been the subject of several case studies in the application of formal methods to real-world software. In her master’s thesis, Endrawaty [End05] modelled the same subsystem of an earlier Fiasco version. She used Promela as specification language and the SPIN model checker [HPV00] to perform simulations and to verify some simple properties. Annamalai [Ann05] extended Endrawaty’s model by adding timeouts among other things, and proved more properties, some of which were liveness properties. Instead of having a lightweight approach to concurrency, they run complete threads in parallel in the model checker leading to huge state spaces. Modelling only two threads where each does only 1 IPC, proving a property took about 8 hours, 2GBs of RAM and 8GBs of hard disk. Proving properties about several IPCs or more than two threads was unfeasible. None of these studies found any error in the code. The bug that we found was only introduced later, when René Reussner rewrote Fiasco’s IPC in his master thesis [Reu05].

Kolanski and Klein worked closely with the L4 development team to obtain a formalization of the kernel’s application programming interface (API) using the B method [KK06]. Concurrency is modeled using B’s parallel composition, hence it is not explicit in their abstract model.

One of the authors of this paper was involved in both the VFiasco and the Robin projects [HT05,TVW09,Tew07]. In both projects the verification of operating-system kernels was attempted, for VFiasco it was the Fiasco microkernel, for Robin it was the Nova micro-hypervisor. At the time of the VFiasco project the Fiasco microkernel was fully preemptable. The Nova micro-hypervisor consists of atomic code blocks with preemption points in between. Both projects concentrated on the modelling and the semantics of certain aspects of the execution environment of these kernels. The verification of larger portions of code was not attempted. Therefore no solution on how to deal with parallelism has been developed in these two projects.

The l4.verified project [Kle09,EKE08,CKS08,Tuc09] attempts the verification of the seL4 kernel. While l4.verified has good chances to finish the first complete verification of a realistic operating-system kernel, we are not aware of any published information about the interruptability of the seL4 kernel or the treatment of parallelism in the verification.

Coyotos [SDSM04] is a secure, microkernel-based operating system built in a new systems programming language (BitC) with a well-defined, mechanically-specified semantics. Singularity [HLA⁺05] is a research operating system at Microsoft Research that aims to build a dependable operating system written in a type-safe language like C# and specified in Sing#, a Spec# extension. These projects are far more comprehensive and long term than our case study.

The Verisoft project [AHL⁺08,DDB08,HP08] aims at the complete verification of a computer system from an e-mail client down to the gate level of the processor. For the verification of their ATAPI disk driver the Verisoft project

used a model in which processor steps are interleaved with the steps of the ATAPI device. To simplify the reasoning the interleaved steps are reordered into larger non-interleaved chunks as much as possible.

8 Conclusions

This work presented a lightweight approach to model concurrency which avoids the need of setting up an interleaving semantics and allows one to reason in a non-parallel fashion. This technique is best suited for systems where a component can be affected by its environment at specific points and by well identified actions.

This approach was applied in the modelling of the IPC subsystem of Fiasco microkernel. It enabled proving some properties of the model with reasonable effort. Under the assumption that our high-level model is faithful and that the identified list of actions is exhaustive, we can ensure that the code honours the properties here studied. During this process we spotted a programming error that, due to its concurrent nature, was hard to be found by testing techniques.

Acknowledgements We would like to thank the operating-systems group at TU Dresden for their support, in particular Rene Reussner and Michael Hohmuth for answering many questions about IPC in Fiasco.

References

- [AHL⁺08] E. Alkassar, M.A. Hillebrand, D. Leinenbach, N.W. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In N. Shankar and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 5295 of *LNCS*, pages 209–224, Toronto, 2008. Springer.
- [Ann05] S. Annamalai. Verification of the Fiasco IPC implementation. Master’s thesis, Dresden University of Technology, December 2005.
- [CKS08] D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer.
- [DDB08] M. Daum, J. Dörrenbächer, and S. Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In B. Beckert and G. Klein, editors, *5th International Verification Workshop*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, 2008.
- [EKE08] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *Verified Software: Theories, Tools, Experiments*, volume 5295 of *LNCS*, Toronto, Canada, Oct 2008. Springer.
- [End05] Endrawaty. Verification of the Fiasco IPC Implementation. Master’s thesis, Dresden University of Technology, March 2005.
- [HHW98] H. Hartig, M. Hohmuth, and J. Wolter. Taming linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART ’98)*, 1998.

- [HLA⁺05] G. Hunt, J.R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B.D. Zill. An overview of the Singularity project. Technical report, Microsoft Research, October 2005.
- [Hoh98] M. Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD-FI98-12, TU Dresden, 1998. available at <http://os.inf.tu-dresden.de/fiasco/doc.html>.
- [Hoh01] Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 217–230, Berkeley, CA, USA, 2001. USENIX Association.
- [Hoh02] M. Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, TU Dresden, Fakultät Informatik, September 2002.
- [HP01] M. Hohmuth and M. Peter. Helping in a multiprocessor environment. In *Proceeding of the Second Workshop on Common Microkernel System Platforms*, 2001.
- [HP08] M.A. Hillebrand and W.J. Paul. On the architecture of system verification environments. In K. Yorav, editor, *Hardware and Software: Verification and Testing, Third International Haifa Verification Conference*, volume 4899 of *LNCS*, pages 153–168. Springer, 2008.
- [HPV00] K. Havelund, J. Penix, and W. Visser. SPIN model checking and software verification, 7th international SPIN workshop. In *SPIN*, volume 1885 of *LNCS*. Springer, September 2000.
- [HT05] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, Glasgow, 2005.
- [KK06] R. Kolanski and G. Klein. Formalising the L4 microkernel API. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theory Symposium*, pages 53–68, Darlinghurst, Australia, 2006.
- [Kle09] G. Klein. Operating system verification—an overview. *Sādhanā*, 34(1):27–69, Feb 2009.
- [OSRS01] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.
- [Reu05] R. Reusner. Implementierung eines Echtzeit-IPC-Pfades mit Unterbrechungspunkten für L4/Fiasco. Master’s thesis, TU Dresden, July 2005.
- [Sch07] E.G.H. Schierboom. Verification of the Fiasco IPC Implementation. Master’s thesis, Radboud University, Computing Science Department, 2007.
- [SDSM04] J. Shapiro, M. Doerrie, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *Proc. NICTA OS Verification Workshop 2004*, Sydney, New South Wales, Australia, October 2004.
- [Tew07] H. Tews. Formal Methods in the Robin project: Specification and verification of the Nova microhypervisor. In H. Tews, editor, *Proceedings of the C/C++ Verification Workshop*, pages 59–68, July 2007. Technical report ICIS-R07015, Radboud University Nijmegen.
- [Tuc09] H. Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning: Special Issue on Operating System Verification*, page 59, 2009. to appear.
- [TVW09] H. Tews, M. Völpl, and T. Weber. Formal memory models for the verification of low-level operating-system code. *Journal of Automated Reasoning*, 42(2-4):189–227, 2009.