

SAME 2005 Forum

Session 2: METHOD ASSERTION

MICROKERNEL API FORMAL MODELISATION

Roger Solà, ENST
Sophie Coudert, ENST

Sophia-Antipolis, France

Sophie Gabriele, STMicroelectronics
Sarah Hoffmann, STMicroelectronics

Rousset, France

Lilian Burdy, Clearsy

[Aix](#) En Provence, France

Abstract:

Any system that needs to be robust and reliable has to rely on the soundness and correctness of its operating system. While careful writing of the code can prevent many bugs, only formal verification can prove the correctness of the system. Traditional operating systems are too complex for current formal analysis tools, but microkernels have a size that makes such a task feasible.

The following paper shows our approach to formalize the L4 microkernel specification. We present a way to transform its informal API description into a model of the formal language B and describe the properties which can be proven in such a model.

1. Introduction

Microkernels have been developed to minimize the size of software that needs to run in privileged CPU mode. They provide a minimal set of general hardware abstractions, which then can be used to implement an operating system with a high level of reliability and security on top. L4[1] is a microkernel based on the principles of minimalism, flexibility, and efficiency. Its small size (about 15,000 lines of C++) and its relevance for security makes it a good candidate for formal analysis. This paper describes our approach to develop a formal model of the API of the L4 microkernel. The goal was to evaluate the possibility to model such software with formal techniques, moreover, the objectives were:

- to describe precisely the mechanisms of L4,

- to obtain a more extensive documentation of the microkernel,
- to highlight the points where the specification was incomplete,
- to prove some static properties on the kernel.

The formalism used to model the system is the Event B formalism[5]. Event B allows to describe a system through a set of data with properties and the events modifying that data: an event either describes an internal behavior of the system or how the system reacts to external actions. Finally, B provides automatic and interactive tools to help proving formally that each event always respects the properties.

2. L4 microkernel family

L4 is the name of a family of second-generation microkernels. There exist various specification versions and implementations. We based our model on the most recent specification version X.2[2] and its implementation *Pistachio* from the University of Karlsruhe.

2.1 Basic concepts of L4

As a microkernel L4 aims to avoid the implementation of any system policies inside the kernel. It focuses on the basic mechanisms that allow to construct a complete operating system in non-privileged mode on top of L4. Its three fundamental abstractions are: address spaces, threads, and inter-process communication (IPC).

First of all, L4 provides *tasks*, that is, virtual address spaces. The address space translation towards physical memory is established decentralized outside the kernel. This works as follows: Initially, *sigma0*, the task created first, finds the available physical memory in its address space. It can share this memory with other tasks by sending (*mapping*) them one or more pages of virtual memory. The receiving task specifies the place where to put the pages and the kernel establishes the new mapping by associating the virtual pages of the receiving task with the same physical memory that is associated with the pages that have been sent. The task can then access the memory or map it recursively on to other tasks. Page faults are forwarded to a specific thread, the *pager*, which has to resolve the fault by providing a mapping.

Mappings may be revoked (*unmapped*) at any time, which results in a recursive unmapping of all further mappings that have been established from that page. To do so, the kernel needs to track all mappings in what is conventionally called the *mapping database*.

Threads are execution contexts. As such they are attached to a task and are scheduled by the kernel according to the parameters that can be changed through system calls.

In L4, IPC messages are always sent synchronously and unbuffered. The content of these messages may include simple registers contents and memory regions that are copied directly into the receiving address space. Memory pages are sent to other tasks via IPC as well.

In order to be able to construct a well-confined system on task level, L4 introduces the notion of privileged tasks. All system calls that may influence the state of other tasks or their attached resources can only be used by, *sigma0* and *root*, the only tasks that are created by the kernel itself.

2.2 API structure

L4 is specified by its API. Therefore, it defines the above described abstractions not directly, but only through the twelve different system calls that manipulate them. For example, a thread is represented in the kernel implementation by a thread control block (TCB) which contains all the information concerning the state of the thread: its stack pointer, its pager, the CPU it is running on and so on. In the API, different system calls are defined to interact with the thread, to modify some information in its TCB and, most importantly, to create or delete a thread.

The API is divided in general, architecture-specific and implementation-defined parts in order to allow performance-optimized implementations of L4. This made it difficult to stay at the API abstraction

level without considering existing implementations. Modeling only the general part of the API would have left important parts of the model unspecified, for example, the behavior of the unmapping function. Therefore, we decided to add the implementation-defined parts. For this we used the Pistachio kernel as a reference, but we took care not to model the Pistachio implementation itself.

3. Event B formalism

First of all, an event system comprises a state which is defined using constants and variables. In practice, these constants and variables are mainly simple mathematical objects: sets, binary relations, functions, numbers, etc. Moreover, they can be constrained by some conditions expressing the invariant properties of the system.

Besides its state, an event system contains a number of events which show the way it may evolve. Each event is composed of a guard and an action. The guard is the necessary condition under which the event may occur. The action determines the way in which the state variables evolve when the event occurs.

Events are atomic. When the guards of several events hold simultaneously then at most one of them may be observed.

Once a system is built, one must prove that it is consistent. This is done by proving that each event of the system preserves the invariants. More precisely, it must be proven that the action associated to each event modifies the state variables in such a way that the new state satisfies the invariant under the hypothesis of the former invariant and of the guard of the event.

4. Model of the L4 API

We have constructed a representation of the L4 microkernel API using this event B formalism. We have developed an abstract model of the data inside the microkernel and described the different ways to interact with it. Those interactions are, in fact, the different system calls. In L4, a system call is an entry point that allows carrying out many different actions. We have split those different actions into different events. This allowed us to define more clearly what is necessary to perform a single action and what this action will modify in the kernel.

We have obtained a model describing all system calls and a manual outlining:

- the structure and the properties of the system abstractions

- and for each possible action its prerequisites as well as the modifications carried out by this action.

4.1 L4 data structure abstraction

The API description of the system calls implicitly relies on a kernel state. The B model makes the state explicit by means of a set of variables and constants, and their properties. Thus it provides an abstract view of the kernel structures. Because the API does not really describe those abstractions but only system calls, system call modelisation is guiding the modelisation: when a system call needs or modifies a variable, this variable is added to the model, and, moreover, when a system call needs or adds a property, it is also added to the model. By this means, the data is exactly described at the required abstraction level for specifying kernel behavior according to the API without considering any implementation details.

The model contains an abstract representation of this data and the thereby defined state can be seen as structured by the basic L4 concepts. Each of these concepts is completely represented with its associated information. The most important data structures are threads, address spaces and those used for IPC. We shortly illustrate the approach on threads before giving some considerations about address spaces and IPC.

Among other operations, a thread can be created, executed and deleted. The model comprises a constant set `threads` containing all threads that exist, have existed or will exist in the system. There are set variables classifying threads according to their status: “existing” meaning created and not yet deleted, “to_active” meaning created but not yet executing and “active” meaning executing. In the specification, the following lines describe the set of threads where POW denotes the powerset:

```
SETS
  Threads

VARIABLES
  existing, to_active, active

INVARIANT
  existing ∈ POW(threads) &
  active ∈ POW(existing) &
  to_active ∈ POW(existing - active)
```

Threads that have not yet been created or that have already been deleted are in “threads - existing” (the set of all threads minus the set of existing threads). Furthermore, an active thread has some information associated: an Instruction Pointer (IP), a Stack Pointer (SP), a priority, a pager, and the initialized address space in which it resides. Not all of this information is meaningful for inactive threads. All that information is modeled with sets and functions. For instance, the stack

pointer is modeled as a total function from active threads to integers with type word. Thus, we have the following lines, where $A \rightarrow B$ denotes the set of all total functions from A to B:

```
VARIABLES
  sp, pager, space

INVARIANT
  sp ∈ active → word_t &
  pager ∈ active ∪ to_active → existing
  space ∈ existing → allocated_space
```

The same kind of model has been constructed to describe address spaces and their associated information. `allocated_space` from the example above is a set variable of this model. It is the set of all those address spaces that have already been created but not yet deleted. They are not necessarily initialized.

Modeling the content of the virtual memory was less straightforward. The model considers the basic memory pages of L4 as smallest memory unit. Address spaces are defined as a sequence of pages. Parts of address spaces can be mapped to parts of other address spaces. These mappings are modeled as sets of pairs of pages. The set of all current mappings constitutes the mapping database of our model. The following lines are extracted from the specification and give an idea of how it is specified. Here, $iseq(X)$ denotes the set of all sequences on the set X, i.e. functions from an initial segment I of \mathbf{N} to X seen as sets of couples in $I \times X$; $ran(f)$ denotes the image of the function or relation f, i.e. if f is a sequence, the members of X belonging to it.

```
SETS
  pages

ABSTRACT CONSTANTS
  spaces

ABSTRACT VARIABLES
  mappingDB

INVARIANT
  Spaces ⊆ iseq(pages) &
  ∀(s1, s2) ∈ space2
    (s1 ≠ s2 => ran(s1) ∩ ran(s2) = ∅) &
  ... &
  mappingDB ∈ POW(POW(pages × pages))
```

To complete the model, we have added some other state variables to the model to distinguish temporal states in addition to those corresponding to the main structures of L4. This was often required for the description of IPC. The most prominent example is the variable `thread_state`, which expresses the status of a thread with respect to IPC communication. It indicates whether the thread is sending, waiting for a message... The following lines characterize this variable:

```
ABSTRACT CONSTANTS
  thread_states = 1..11 &
  running = 1 & waiting_forever = 2 &
  waiting_timeout = 3 & ... & dead = 11
```

```
ABSTRACT_VARIABLES
  thread_state
```

```
INVARIANTS
  thread_state ∈ active → thread_states
```

4.2 L4 system call modelisation

Every L4 system call allows to perform several actions. In fact, the API defines only twelve system calls while we identified about 60 actions. Which action is carried out depends on the input parameters of the system call. For example, with `ThreadControl` one may create or delete threads, modify pager, address space, version bits, or scheduler of a thread, or assign interrupt handlers. What is done depends on whether the thread and the address space passed as parameters already exist or not.

Modeling each system call in one single B event would have led to very complex events. Therefore, we have chosen to model each of the possible actions in a separate event. This has the major benefit to describe properly each action in the model with its prerequisites and the modifications it performs on the state of the system. To ensure the consistency and completeness of each system call splitting, input parameters configurations have been exhaustively classified into exclusive cases. For each resulting action (event in B), the conditions on the parameters of the originating system call appear as a guarded precondition for the event to occur. The above mentioned system call `ThreadControl` is described with B events such as thread deletion, address space creation, or scheduler modification. The latter is illustrated below:

```
ThreadControl_modify_scheduler =
  ANY
    in_scheduler, thread

  WHERE
    in_scheduler ∈ existing &
    space(thread) ∈ {root, sigma0} &
    thread ∈ existing

  THEN
    scheduler(thread) := in_scheduler
END ;
```

Here, if `in_scheduler` and `thread` both exist, then `in_scheduler` can be assigned to `thread` provided that `thread` resides in an address space allowing this operation, i.e in a privileged address space.

Thus, every system call is described by a set of events that use input parameters and the system state to generate output parameters and modifications of this state (address spaces, threads, etc.). The system state is entirely described by the L4 abstraction model explained in the previous section.

We have modeled ten of the twelve system calls. In the course of this work the API abstraction level appeared to be a little too concrete to allow simple specifications and proofs of reasonable size. For example, the mapping database is described with a non-obvious abstract set-theoretic structure, so the mapping operations have been modeled at this abstract level, considering as parameters set of pages to map and not set of addresses in the address space.

The final version of the model is slightly more abstract than the API. Another example is that it directly considers threads and not thread identifiers any more. At this level, all has been reasonably specified and proven.

4.3 Model translation into a manual

The AtelierB toolkit allows to create a translation of the B model into informal language. This *manual* constitutes another way to represent the L4 API. Each event is presented with its guarded preconditions and its modifications. Moreover, different representations of the model are constructed, most notably, a dependency graph between variables, events that read them, and events that modify them.

For the example of the scheduler modification with the `ThreadControl` system call above, we obtain as description for the event:

```
Event modify_scheduler implemented by
ThreadControl system call:
Input parameters :
  - the new scheduler to set
  - a thread
When
  the new scheduler is an existing
  thread and
  the thread is privileged and
  the thread is an existing thread
Then
  the scheduler of the thread is set
  to the new scheduler.
```

The AtelierB toolkit creates the manual automatically. The user « only » has to give the translation of all B constructs he has used to build his model. The result is a hyper-linked document that allows navigating through the model. It offers a different view on the API. Instead of regarding it from a system call point of view, it represents it from the point of view of the actions that can be performed by the kernel. This description is informal but built from the formal model.

4.4 Static properties of the kernel

As a last aspect of the modeling, we have proven some static properties in the kernel. For the L4 API, it has been difficult to identify meaningful properties since the L4 kernel on its own is a passive microkernel that does not protect itself against misuse. The L4 kernel developers argue that it is the task of the privileged tasks to protect

themselves and the system built on top of the kernel against misuse of the kernel system calls. Thus, we have only defined and proven very basic properties concerning the coherence of the abstractions. Regarding the example of the TCB from section 4.1, we delivered properties such as "every active thread has a pager" or "a global thread ID is associated to at most one thread".

In the more abstract model that we have eventually constructed we have added more properties. For instance, describing the pager of a thread with a thread and not with a thread id allows to express in an easier way properties about the state of the pager of a thread. We have also considered consistency properties that are not described in the API but that are good properties if one wants to construct a complete operating system on top of the microkernel. For instance, we have added a property expressing that the pager of an active thread is never unspecified: to prove it, we have also strengthened the guard of the event modifying the pager requiring that the given pager parameter is not zero.

5. Related work

The project which is most closely related is the L4 Kernel Verification Project of NICTA[4]. There also a formal model of a part of the L4 specification has been developed. Their goal is to arrive in a top-down approach at a formalization of the Pistachio implementation of L4. Therefore, the model is more hardware-oriented than the one presented in this paper.

The VFiasco project[3] aims at verifying security-relevant properties of the L4 implementation Fiasco. For this, it favors a bottom-up approach through an automated formalization of the C++ source code to prove the desired properties. Our approach is different from both projects: we did not work on a particular implementation but we have constructed an abstract model with some reasonable properties of the microkernel in order to build a non-ambiguous documentation.

6. Conclusion

To conclude, we consider that the development of this model allowed us to obtain more confidence in the correctness of the API, to describe its behavior from a different point of view, and to establish a deeper understanding on how it works. We did not find any critical errors or inconsistencies in the API document but many points where we proposed to rewrite, to explain or to complete certain paragraphs. Finally, it has to be mentioned that this work was mostly realized without any prior deep knowledge of B or L4. This demonstrates that it is

realistic to construct a formal model and to benefit from it in short term.

For the future, the model needs to be completed. Apart from the two missing system calls there are some aspects of the L4 API which could help make the model more accurate: the API defines protocols for page faults, interruption, and scheduling. Some of them we already included in the model, some are yet to do. Another improvement would be to extend the model to a multi-processor environment.

Using the kernel model, we may also model simple L4 tasks. There, especially the privileged tasks, `sigma0` and `root`, are of interest because only with them we can obtain a system where we can prove more complex properties.

References

- [1] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No.1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.
- [2] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: <http://l4hq.org/docs/manuals/>.
- [3] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD–FI02–03–März 2002, Dresden University of Technology, 2002. Available from URL: <http://os.inf.tu-dresden.de/vfiasco/>.
- [4] H. Tuch and G. Klein. Verifying the L4 Virtual Memory Subsystem. Proc. NICTA Formal Methods Workshop on Operating Systems Verification, Technical Report 04011005T-1, National ICT Australia, 2004
- [5] Jean-Raymond Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. FME 2003, pages 51-74.