

# The B-Method for the Construction of Microkernel-Based Systems

Sarah Hoffmann, STMicroelectronics  
Germain Haugou, STMicroelectronics  
Sophie Gabriele, STMicroelectronics  
Lilian Burdy, ClearSy

## Introduction

Microkernels have been developed to minimize the size of software that needs to run in privileged CPU-Mode. They provide only a 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 is a second generation microkernel based on the principles of minimalism, flexibility and efficiency. Its small size (about 15,000 lines of C++) and its relevance for security make 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. 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.

## Basic concepts of L4

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

Tasks provide virtual address spaces. The address space translation towards physical memory is established decentralized by the tasks themselves. 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 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. 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. For this, 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.

IPC allows to send messages synchronously and unbuffered between two threads. The content of these messages may consists of simple registers contents and of memory regions that are copied directly into the receiving address space. Memory pages are sent to other tasks via IPC as well.

## 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 allows defining 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 the system calls and a manual outlining:

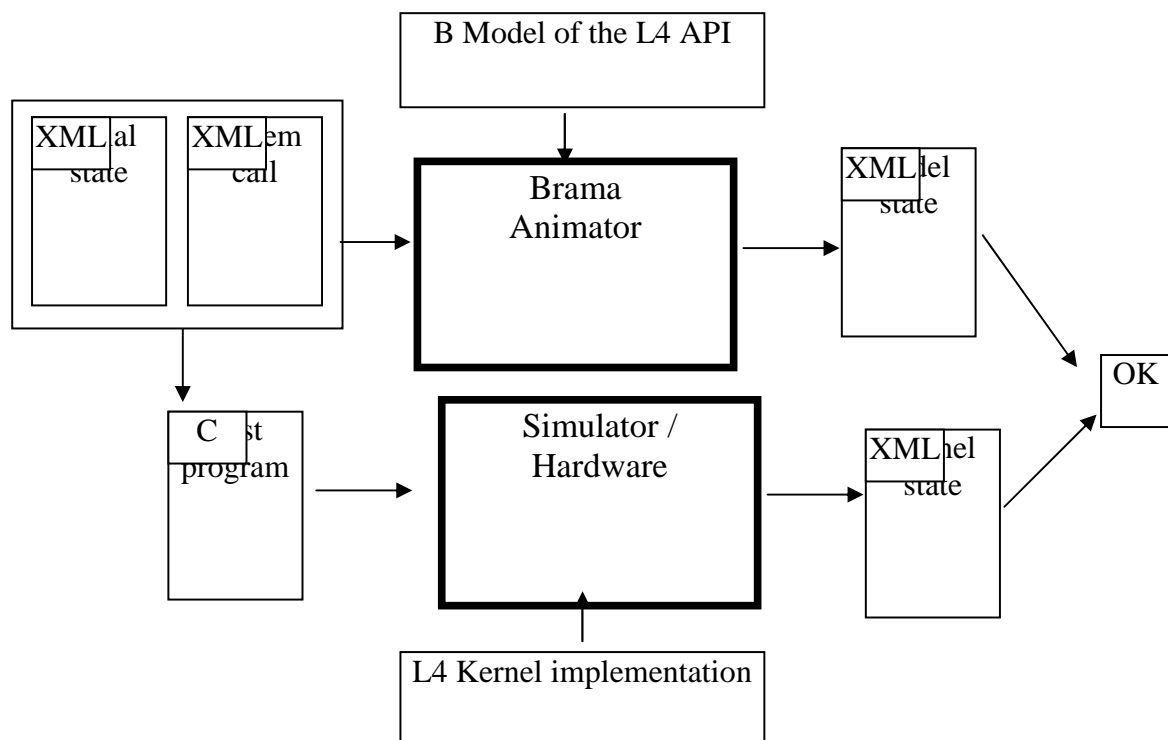
- the structure and the properties of the system abstractions
- and for each possible action its prerequisites as well as the modifications carried out by this action.

The data structures mainly represent the memory and the threads. These structures are not described in the API, but they appear through the specification of the system calls. In the model, they correspond to the definition of constants and variables.

On the other hand, each action is guarded by a predicate on those variables defining the conditions on which the corresponding system call can be called. This guard depends also on the system call parameters.

## Animation and test

To validate the model against the running code (and vice versa), we have used the Brama animator framework in order to test the code using the model as an oracle. A test case consists of a kernel pre-state and a system call with specific parameters. The corresponding test application first makes some system calls to position the kernel in the pre-state, then it executes the system call, and finally the kernel post-state is dumped into a file. Using the animator, one can also set the model in a given state, execute some events and dump the model state into a file.



We have developed an interface that allows to convert a kernel state into a model state, associating the model variables with the implementation data. We have also converted a system call into a sequence of event executions. This allows to put the kernel and the model in an equivalent state starting from a unique description, to execute the system call and to compare the two resulting states in order to validate the conformance of the code to the model.

## Conclusion

To conclude, we consider that the development of this model allows 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. The test framework allows to validate the model against an implementation; the relation between the model and the implementation is not proven (as it could be in a complete formal development with code generation) but the test allows to ensure more confidence. For software where efficiency is such a crucial requirement that it rules out code generation, this kind of test framework can be a realistic solution to construct a validation at two levels: at specification level, when constructing the model, and at implementation level when testing it.

## References

- 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.
- 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/>.
- 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/>.
- 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