Verifying the PikeOS Microkernel: First Results in the Verisoft XT Avionics Project

Christoph Baumann[1] and Thorsten Bormer[2]

[1] Saarland University, Dept. of Computer Science, Saarbrücken, Germany
baumann@wjpserver.cs.uni-sb.de
[2] University of Koblenz, Dept. of Computer Science, Koblenz, Germany
tbormer@uni-koblenz.de

Abstract. In this paper, we are giving an overview of the ongoing Verisoft XT Avionics project reporting on the progress of the project, and presenting first results in the verification of the system calls of the microkernel.

The goal of Verisoft XT Avionics is to formally verify an existing operating system which has not been designed for deductive code verification. The system under consideration is PikeOS, a state-of-the-art microkernel developed by SYSGO AG, which is used in a variety of embedded applications. For automated formal verification we deploy Microsoft's Verifying C Compiler (VCC).

Introduction Functional correctness of the built-in operating system is a crucial requirement for the reliability of safety- and security-critical systems. Hence, operating system kernels are a worthwhile target for formal verification. The goal of the Verisoft XT Avionics sub-project is to prove functional correctness of the microkernel in PikeOS, a commercial operating system for embedded systems [1].

Unlike the predecessor project Verisoft I, we do not target pervasive verification of the whole system stack consisting of – amongst others – the hardware, compilers and the microkernel. We rather pick one layer, the PikeOS microkernel, to show that verification of a real world implementation of a considerable size is a feasible task, taking advantage of the high degree of automation when using verification tools like VCC.

The PikeOS System PikeOS (see `http://www.pikeos.com/`) consists of a microkernel acting as paravirtualizing hypervisor and a system software component. The verification target we have chosen for our project is the PikeOS version for the PowerPC processor family, the OEA architecture, and the MPC5200 platform [6,7].

The roots of PikeOS can be seen in the L4 family of microkernels, although the PikeOS kernel is particularly tailored to the context of embedded systems, featuring real-time functionality and resource partitioning.

Along with another component on top of the microkernel layers, the so-called system software, the PikeOS kernel provides partitioning features. This allows to virtualize several applications on one CPU, where each application runs in a secure environment with configurable access to other partitions, if desired.

In order to provide real-time functionality, there are many regions within the kernel code where execution may be preempted – we thus have a concurrent kernel. Moreover, the kernel is multi-threaded.

To allow for easy adaptation to other platforms and CPU families, the kernel is structured into three layers. There are two layers close to the hardware providing processor abstraction as well as platform-dependent functions. These layers are partly written in PowerPC assembly, which makes about a quarter of the overall code. On top of these abstraction layers, the generic microkernel provides features like tasks and threads with real-time scheduling and memory management and is written entirely in C. This layer is approx. the size of both lower layers combined. In total, the code size of PikeOS is smaller than that of, e.g. Linux by several orders of magnitude.

Verification Methodology and Toolchain To show correctness of the implementation of PikeOS, we use the Verifying C Compiler (VCC) developed by Microsoft Research. The VCC toolchain allows for modular verification of C programs using method contracts and invariants over data structures. These contracts and invariants are stored as annotations within the source code, similar to the approach and syntax used in, e.g. Caduceus [5] or SPEC# (see `http://research.microsoft.com/specsharp`).

As most verifying compilers today, VCC works using an internal two-stage process. Firstly, the annotated C code is translated into first-order logic via an intermediate language called BoogiePL [4]. BoogiePL is a simple imperative language with embedded assertions. This representation is further transformed into a set of first-order logic formulas (called verification conditions), which state that the program satisfies the embedded assertions. In the second stage, the resulting formulas are given to an automatic theorem prover (TP) resp. SMT solver (in our case Z3 [3]) together with a background theory capturing the semantics of C's built-in operators, etc. The prover checks whether the verification conditions are entailed by the background theory. Entailment implies that the original program is correct w.r.t. its specification.

One distinguishing feature of this verification tool, especially important in our setup, is the support for concurrency (see [2] for details).

Verification of the PikeOS Microkernel As already mentioned, we are verifying a real-world implementation of a microkernel that is deployed and used in industry and has not been written with verification in mind. For now, we only consider functional properties of the microkernel – the specifications for this are in parts derived from the informal descriptions of the PikeOS kernel (including the reference manual). In addition, code analysis and inspection provides further insight into implementation details.

One of the first steps to verify PikeOS have been helper functions with limited dependencies on other parts of the system. The next verification target are system calls, both to demonstrate pervasive verification through the different layers of the microkernel and to introduce externally visible contracts of the kernel.

Verification of System Calls As a first target for verification, we have chosen a simple system call which changes the priority of a thread (named `p4syscall_fast_set_prio`). This call has a rather simple functionality, but it serves very well as an example because its execution spans all levels of the PikeOS microkernel, from high-level kernel functionality to hardware-related levels and the user-level interface (system calls are invoked via user interrupts).

Handling Assembly Code On the hardware-related level of the system call to be verified, (inline) assembly code is used to directly access the underlying hardware. We model the relevant parts of the hardware as a C struct in the ghost-state of the program, to be able to verify code including PowerPC assembly. The effects of assembly instructions can then be expressed using this ghost model.

Henceforth, each assembly instruction is modeled by a corresponding specification function that operates on the machine model C struct – this allows us to replace assembly code by a sequence of specification functions being able to be annotated and verified using VCC like normal C code. The specifications of these assembly code parts are then used as contracts on the upper layers of the system call implementation.

The Abstract Layer On the upper layers of the kernel, the functional verification of PikeOS has to establish the contracts that enable users of the microkernel to depend on the specifications made for the outer boundaries of the kernel as given in its documentation.

Because system calls are at the user's interface to the kernel and the PikeOS system is multi-platform, the kernel's specification has to hide any PowerPC implementation details to ensure proper encapsulation. Further, the specification has to reflect the effects of the system calls as given in the documentation of PikeOS and thus the entities defined in this documentation have to be available on the VCC specification level.

Therefore, an abstract model of the kernel's state is introduced that captures the essential properties of the kernels state, including the current hardware configuration. As with the model of the PowerPC hardware, this abstract model is kept in the ghost state of the program.

Some of the fields of the abstract kernel model are related to the underlying machine state and thus to the hardware model mentioned above. These relations are explicitly specified with object invariants depending on both the abstract model and the hardware model. The VCC methodology then ensures that the relation between the abstract model and concrete machine model is obeyed by the implementation.

First Results Based on the model of the underlying PowerPC hardware, we were able to verify low-level functions of the PikeOS kernel. This then enabled us to verify first system calls of the kernel – for the time being under the restriction that no preemption takes place during the call. In the verification of a system call that crossed the boundary between C and assembly, a first version of an abstract model of the kernel's state was defined. The elements of this abstract state that are visible to other components directly correspond to the entities described in the PikeOS documentation.

In the next steps, we will extend the verification effort to further, more complex, system calls. In addition, annotations dealing with concurrency and ownership are ongoing work.

References

1. Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Better avionics software reliability by code verification. In *Proceedings, embedded world Conference, Nuremberg, Germany*, 2009.
2. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs, 2008.
3. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 14th International Conference, Budapest, Hungary*, LNCS 4963, pages 337–340. Springer, 2008.
4. Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
5. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.
6. Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC^{TM} Architecture*, 3rd edition, September 2005.
7. Freescale Semiconductor. *MPC5200B User's Manual, Rev. 1.3*, Sep 2006.