Lecture Note
seL4: Formal Verification of an OS Kernel

Chun-Kun ”Amos” Wang

1 Abstract

Complete formal verification is the only known way to guarantee that a system is free of programming errors. They present their experience in performing the formal, machine-checked verification of the seL4 microkernel from an abstract specification down to its C implementation. According to this paper, this is the first formal proof of functional correctness of a complete, general-purpose operating-system kernel.

- Assume the correctness of the compiler, assembly code, boot code, management of caches, and the hardware.

2 seL4 Overview

seL4, similarly to projects at Johns Hopkins (Coyotos) and Dresden (Nova), is a third-generation microkernel, and is broadly based on L4 and influenced by EROS. seL4 is a member of the L4 microkernel family, designed to provide the ultimate degree of assurance of functional correctness by machine assisted and machine-checked formal proof.

- It features abstractions for virtual address spaces, threads, interprocess communication (IPC), and, unlike most L4 kernels, capabilities for authorization.

3 Kernel design process

OS developers tend to take a bottom-up approach to kernel design. In contrast, formal methods practitioners tend toward top-down design. This leads to designs based on simple models with a high degree of abstraction from hardware. For both OS developers and formal methods practitioners, it uses the functional programming language Haskell to provide a programming language for OS developers, while at the same time providing an artefact that can be automatically translated into the theorem proving tool and reasoned about as shown in Figure 3.1.

The square boxes are formal artefacts that have a direct role in the proof. The double arrows represent implementation or proof effort, the single arrows represent design/implementation in influence of artefacts on other artefacts. The central artefact is the Haskell prototype of the kernel. The prototype requires the design and implementation of algorithms that manage the low-level hardware details. To execute the Haskell prototype in a near-to-realistic setting, we link it with software (derived from QEMU) that simulates the hardware platform.

We restrict ourselves to a subset of Haskell that can be automatically translated into the language of the theorem prover we use. For instance, we do not make any substantial use of laziness, make only restricted use of type classes, and we prove that all functions terminate.

While the Haskell prototype is an executable model and implementation of the final design, it is not th efinal production kernel. We manually re-implement the model in the C programming language for several reasons.

- the Haskell runtime is a significant body of code.
- the Haskell runtime relies on garbage collection which is unsuitable for real-time environments.
- While an automated translation from Haskell to C would have simplified verification, we would have lost most opportunities to micro-optimise the kernel.

4 Verification

They use the theorem prover Isabelle/HOL. Interactive theorem proving requires human intervention and creativity to construct and guide the proof. However, it has the advantage that it is not constrained to specific properties or finite, feasible state spaces, unlike more automated methods of verification such as static analysis or model checking.

Figure 4.1 shows the specification layers used in the verification of seL4.

- Abstract specification
  - It describes what the system does without saying how it is done, like functional behavior of kernel operations.
  - Figure 4.2 is an example of scheduling. No scheduling policy is defined at the abstract level. Instead, the scheduler is modeled as a function picking any runnable thread that is active in the system or the idle thread.

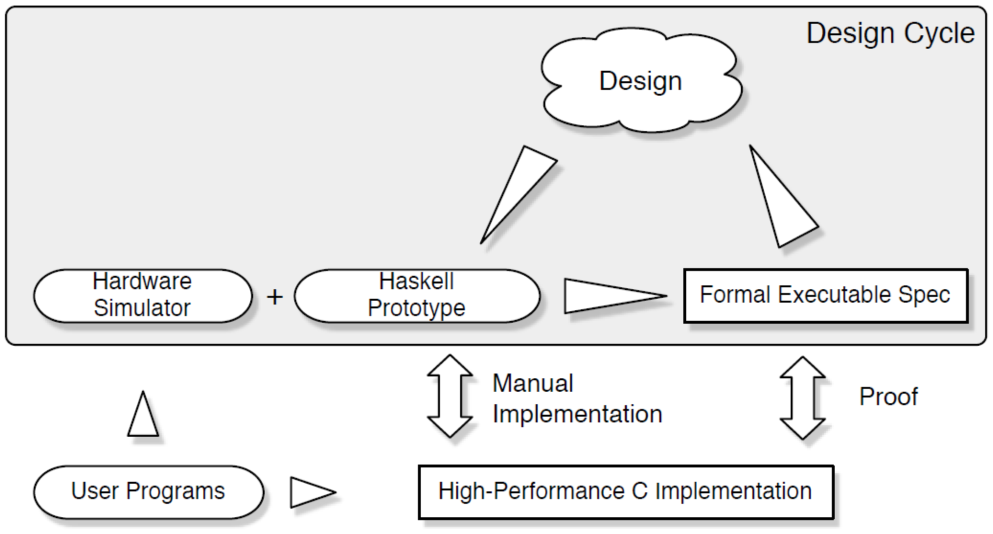

Figure 3.1: The seL4 design process

- We make use of non-determinism in order to leave implementation choices to lower levels.

- Executable specification
  - It is generated from Haskell into the theorem prover and fills in the details left open at the abstract level and to specify how the kernel works as opposed to what it does.
  - The executable specification is deterministic; the only non-determinism left is that of the underlying machine.
- C implementation
  - The most detailed layer in our verification is the C implementation.
  - The translation from C into Isabelle is correctness-critical and we take great care to model the semantics of our C subset precisely and foundationally.
- Machine model
  - Programming in C is not sufficient for implementing a kernel, like assembly.
  - The basis of this formal model of the machine is the internal state of the relevant devices, collected in one record machine_state.

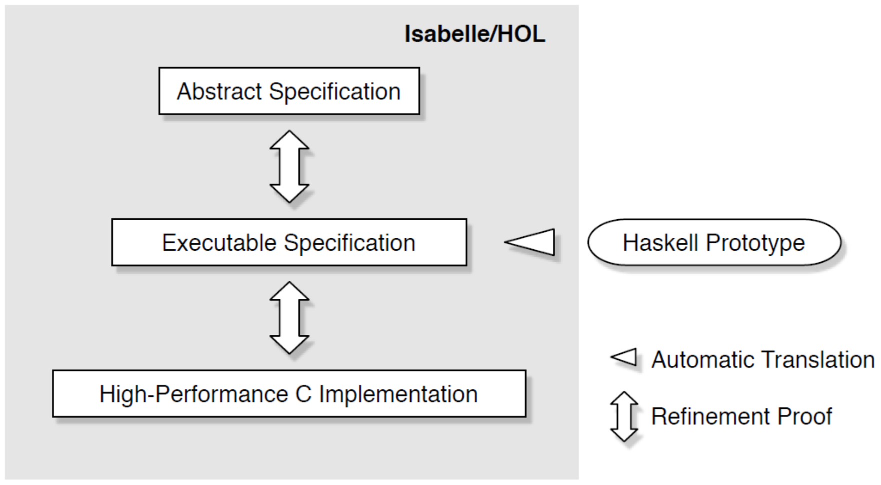

Figure 4.1: The refinement layers in the verification of seL4

```
schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

Figure 4.2: Isabelle/HOL code for scheduler at abstract level

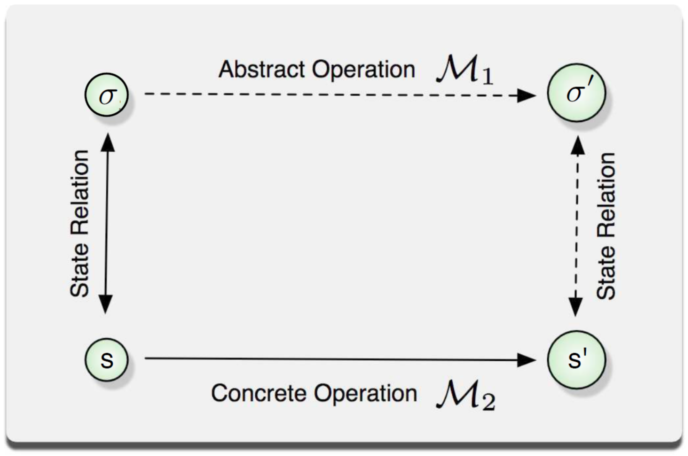
Abstract Operation $\mathcal{M}_1$
σ
σ'
State Relation
State Relation
s
s'
Concrete Operation $\mathcal{M}_2$

Figure 5.1: Forward Simulation

5 Proof

The main property we are interested in is functional correctness, which we prove by showing formal refinement. We have also proved the well-known reduction of refinement to forward simulation, illustrated in Figure 5.1. To show that a concrete state machine M_2 refines an abstract one M_1, it is sufficient to show that for each transition in M_2 that may lead from an initial state s to a set of states s', there exists a corresponding transition on the abstract side from an abstract state σ to a set σ' (they are sets because the machines may be non-deterministic). The transitions correspond if there exists a relation R between the states s and σ such that for each concrete state in s' there is an abstract one in σ' that makes R hold between them again.

Let machine M_A denote the system framework instantiated with the abstract specification, let machine M_E represent the framework instantiated with the executable specification, and let machine M_C stand for the framework instantiated with the C program read into the theorem prover.

- Theorem 1: M_E refines M_A.
- Theorem 2: M_C refines M_E.
- Theorem 3: M_C refines M_A.

6 Results

The overall code statistics are presented in Figure 6.1. The project was conducted in three phases.

	Haskell/C	Isabelle	Invariants	Proof
	LOC	LOC		LOP
abst.	—	4,900	$\sim$ 75	110,000
exec.	5,700	13,000	$\sim$ 80	
impl.	8,700	15,000	0	55,000

Figure 6.1: Code and proof statistics

1. An initial kernel with limited functionality (no interrupts, single address space and generic linear page table) was designed and implemented in Haskell, while the verification team mostly worked on the verification framework and generic proof libraries.
2. The verification team developed the abstract spec and performed the first refinement while the development team completed the design, Haskell prototype and C implementation.
3. It consisted of extending the first refinement step to the full kernel and performing the second refinement. The overall size of the proof, including framework, libraries, and generated proofs (not shown in the table) is 200,000 lines of Isabelle script.