

Technical Perspective

The Scalability of CertiKOS

By Andrew W. Appel

FOR MODERATE-SIZE SEQUENTIAL programs, formal verification works—we can build a formal machine-checkable proof that a program is correct, with respect to a formal specification in logic. Machine-checked formal verifications of functional correctness have already been demonstrated for operating-system microkernels, optimizing compilers, cryptographic primitives and protocols, and so on.

But suppose we want to verify a high-performance hypervisor kernel programmed in C, that runs on a real (x86) machine, that is capable of booting up Linux in each of its (hypervisor) guest partitions? Real machines these days are multicore—the hypervisor should provide multicore partitions that can host multicore guests, all protected from each other, but interacting via shared memory synchronized by locks. Furthermore, the operating system itself should be multicore, with fine-grain synchronization—we do not want one global lock guarding all the system calls by all the cores and threads.

The authors of the following paper illustrate that formal verification can scale up to a moderate-size program (6,500 lines of C) that has substantial shared-memory concurrency. They succeed by a ruthless and disciplined use of modularity and *contextual refinement*: each module of the C program behaves “the same” as its functional specification *in any context*; and each module compiles to assembly language that behaves “the same” as the C program *in any context*. They certify this with machine-checkable proofs. Therefore, they call the approach *Certified Abstraction Layers*.

Here’s an example of contextual refinement: Suppose module *A* interfaces to module *B* using the principle of Abstract Data Types (also known as “representation hiding”): Module *B* has private variables with public interface methods. Module *A* never

The authors illustrate that formal verification can scale up to a moderate-size program that has substantial shared-memory concurrency.


reads and writes *B*’s variables directly but calls upon *B*’s methods to do it. We can say that module *A* is the *context* for running module *B*. What can *A* observe about *B*? Only the data values passed to, and returned from, *B*’s interface methods. We can substitute a different implementation for *B* that “behaves the same” from *B*’s point of view. In particular, we could write a *functional specification* for *B* written in a functional programming language or written in mathematical logic; then we could write a C program implementing module *B*. Module *A* cannot tell the difference between the functional specification and the C-language implementation. Then, use a proved-correct C compiler, and module *A* cannot tell the difference between the C program and the assembly-language program.

That explanation works well for single-threaded programs where the modules are connected at function-call interfaces. The CertiKOS team has previously demonstrated their Certified Abstraction Layer methodology to prove the correctness of a single-core version of CertiKOS.

In this paper, the authors describe what it takes to extend the methodology to concurrency.

With multithreading, module *A* (the context) can interface to module *B* not only through function calls, but by acquiring and releasing locks and then reading and writing the shared memory locations controlled by those locks. This gives a more complicated notion of “behaves the same”—it’s not just the arguments and return-values of procedure calls. With multicore, contextual refinement describes, in each thread individually, the relation between a *synchronization trace* of the functional specification and of the C program.

What we all know about software is that it never stays still. It would do no good to verify correctness of an operating system, if next week when we commit a change to the source-code repository, we would have to throw away the proof and start over. Proofs about programs must be highly modular, just like the programs themselves. The CertiKOS team shows how to design contextual refinements that are module-by-module, even in the presence of concurrency. That means, when you commit a change to the implementation of one module, you just need to adjust the proof of that module alone.

To make their proofs so modular, the CertiKOS team has had to pay careful attention to abstraction interfaces. As a result, the C program is extremely well structured, layered, and modular. This has benefits even for those who read the C program without ever looking at the proofs. 

Andrew W. Appel is the Eugene Higgins Professor of Computer Science at Princeton University, Princeton, NJ, USA.

Copyright held by author.