Evaluatable, High-Assurance Microprocessors

David Greve and Matthew Wilding

`{dagreve,mmwildin}@rockwellcollins.com`

Rockwell Collins Advanced Technology Center
Cedar Rapids, IA 52498

Introduction

The CAPS (Collins Adaptive Processing System) is a family of Rockwell Collins proprietary processors. In a multiyear IR&D effort, Rockwell Collins adapted and developed techniques that allow for formal code verification of the microcode running on members of this family. The motivation for this research was that CAPS microprocessors are used in some of the most safety-critical products that Rockwell Collins sells, and current microprocessor verification and certification techniques are extremely laborious. On this project formal verification techniques were demonstrated on examples of actual microcode. Some of this work is publically documented [Greve00a, Wilding01a]. The code verification techniques we developed for critical microcode can be generalized to other kinds of code.

In this summary paper we provide an overview of what we consider to be three of the important aspects of this work: our approach to model development, our approach to code proof decomposition, and several code proof automation techniques that we adapted and/or developed.

Formal Model Development Tools

Rockwell Collins has developed tools and techniques for writing and reasoning about formal models. The models are consistent with the logic supported by the ACL2 theorem prover, and can be reasoned about using that tool. Models developed using this approach execute about as fast as simulators written in conventional, imperative languages like C. Because these formal models are written in a standard programming language, they can be integrated with other software, such as the simulator used for microcode development that is shown in Figure 1.

The tools used in building machine models provide the automatic generation of update and access theorems for each element of the state. In the CAPS project, the underlying model contained 337 state elements. For this model the tools automatically generated thousands of theorems, which were subsequently

processed by the ACL2 theorem prover to ensure their correctness. These theorems were then used in proofs about microcode running on the model.

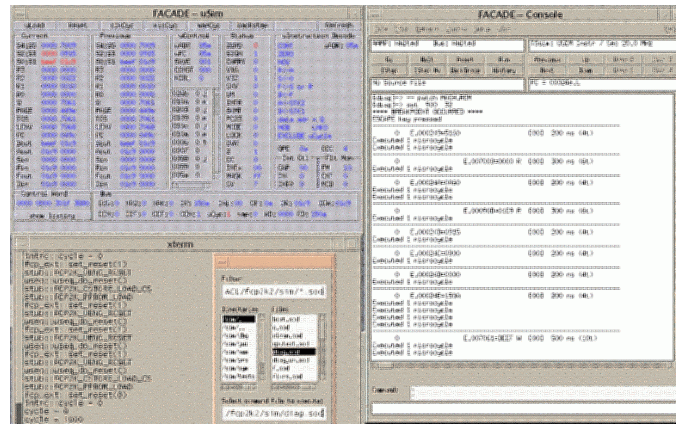


Figure 1 Microarchitecture simulator based on formal model ([Wilding01])

Code Proof Decomposition

A crucial challenge in code correctness proofs is developing a methodology for breaking the proof down into smaller, more manageable pieces. In the CAPS proofs we were faced with figuring out how to prove that some segments of microcode correctly implemented machine instructions. The CAPS model has hundreds of state elements and the symbolic execution of each line of microcode typically involves more than 1,000 symbolic updates to the state. Even the simplest instructions are implemented with multiple lines of microcode, and certain invariants on the state preserved during the execution of microcode sequences are crucial to establishing their correct functioning.

Figure 2 shows the CAPS proof architecture. Broadly speaking, three proof decomposition techniques were exploited in the CAPS program.

1. *Separation of implementation details from algorithm* The correctness of code depends on both the proper implementation of an algorithm and the correctness of an algorithm. A code proof can be broken into lemmas that address these two correctness issues separately. This decomposition is reflected in the CAPS proof architecture: the "single microcode line specs" calculate the result of a single line of CAPS microcode, ignoring the irrelevant aspects of the (very complex) interpreter model of the microarchitecture of the machine.

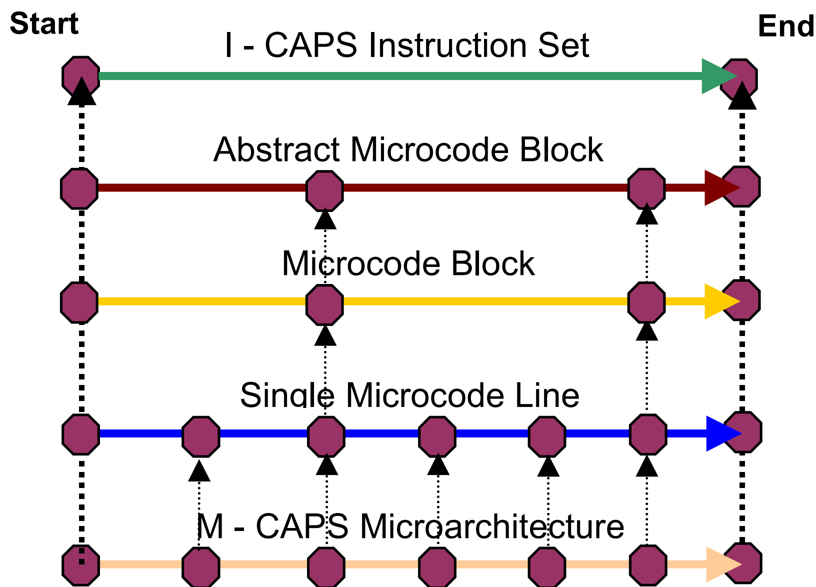
2. *Exploitation of the block structures to break code execution into steps*. Code is organized into blocks and the associated proofs can be decomposed into lemmas about the corresponding steps. This

decomposition is reflected in the combining of single microcode line specs into microcode block specs.

3. *Loop simplification* Proofs involving loops can be further broken into proofs about the correct implementation of the loop (using some kind of recursive function) and a simple specification where the state elements that are manipulated by the loop are specified individually in the body of an otherwise non-recursive specification. This decomposition is reflected in the use of abstract microcode block specs.

An important innovation used in the CAPS code proofs is the use of the underlying machine to describe irrelevant parts of the model. For example, for a particular line of microcode all but a handful of the hundreds of state elements of the microarchitecture model are irrelevant. We specify their behavior in the "single microcode line specs" by using the underlying microarchitecture model. This makes writing the models easy – only a few relevant parts of the state are explicitly modeled – and allows equality reasoning throughout the proofs. This technique is described (in a PVS theorem prover context) in [Wilding97].

Figure 2 CAPS Proof architecture



Modeling and Proof Automation

Modeling and proving programs correct requires ingenuity since intellectual effort is required to understand what a program does. However, many of the tasks associated with the process of modeling and proving code correct can be automated. In fact, automation of the model and proof process is crucial for

making code proofs practical. Although our proof decomposition approach transforms the code proof challenge into smaller challenges, those smaller challenges are still formidable. Rockwell Collins' CAPS microcode verification program included three basic types of automation.

- *Automated tools for generating proofs* These tools generate input for a theorem prover and significantly simplify the task of guiding them to correctness proofs. For CAPS this kind of tool included
    - Several *proof-generating* macros that automate much of the proof decomposition process described in the previous section.
    - A *"reader"* that provides convenient expression of machine models and which generates type rules for manipulating state.
- *Reasoning libraries* Once the proofs are decomposed into smaller pieces there often remain proof obligations involving fundamental machine arithmetic properties. One of the most important instances of code proof automation in the CAPS work is the Super-IHS library, which contains many rules that provide a standard strategy for simplifying expressions that result from symbolic code execution. Super-IHS extends the IHS book that is part of the standard ACL2 distribution.
- *Theorem prover enhancements* In our experience, every application of a theorem prover to an industrial problem results in tool improvements. In the CAPS work several improvements were made to ACL2, most significantly the introduction of stobjs and development of the nu-rewriter, which provides efficient automated simplification of expressions involving state accesses and updates.[1]

Automation of the modeling and proof process in these respects was crucial for demonstrating formal correctness of some of the CAPS microcode. Nearly all of the techniques we developed apply to other kinds of code proofs besides microcode proofs, such as machine code proofs. Furthermore, most also apply to other theorem proving systems besides ACL2. What has emerged from the CAPS project and other previous projects are *ad hoc* approaches for tackling the fundamental challenge of complexity in code proofs using proof decomposition and automation.

---

[1] The nu-rewriter was developed under contract to Rockwell Collins by J Moore of the University of Texas at Austin. The problem Rockwell Collins faced was that the simplification of state access and update terms requires something like outside-in rewriting, and yet most problems benefit from the inside-out simplification approach implemented in ACL2. Dr. Moore's ultimate solution involves a careful integration of a limited form of outside-in rewriting with the standard ACL2 simplification strategies and overcomes this crucial problem in code proof development. The nu-rewriter was recently released (in ACL2 version 2.6) and is documented in [Moore01].

An example ACL2 code proof

A small code proof that illustrates some aspects of the CAPS work is available. The problem is from [Legato00], and the proof can be checked using ACL2 2.6.

The example is in some respects simpler than the kinds of microcode proofs done under the CAPS program.

- The formalization of the underlying machine model is much less complex in this example compared with the CAPS microarchitecture model. There is no interpreter in this example; the functional description of each line of the code is introduced at a level of abstraction similar to the "single microcode line" of Figure 2.
- The block structure of the code is constructed by hand and added as a function, also simplifying the proof burden compared with CAPS proofs.
- The ISA-level of this example is simpler than the microarchitecture level at which the CAPS model is written.

Despite these differences, several aspects of the CAPS work can be observed in this example.

- Although the Super-IHS book is not loaded, the underlying philosophy of that book, which is similar to that of the publically-available IHS book, is evident. Note in particular that, although the machine operations are defined in terms of arithmetic operations on integers, some of the proofs use induction schemes that would normally be associated with operations defined on bit-vectors.
- Although the Rockwell Collins "reader" used for model building is not used in this example, rewrite rules that were generated by the reader on another problem were adapted for use here. Note that we would normally generate these rules automatically.
- This example relies on the nu-rewriter for efficient and automatic simplification of expressions involving state access and update.

Summary

At Rockwell Collins we are learning how to apply theorem provers to safety- and security- critical applications requiring high assurance. The use of automation and proof decomposition is critical to successful use of these tools.

Bibliography

[Greve00a] David Greve, Matthew Wilding, and David Hardin, *High-Speed, Analyzable Simulators*, invited chapter in **Computer-Aided Reasoning: ACL2 Case Studies**, Kluwer Academic Publishers, 2000. (ISBN 0-7923-7849-0)

[Legato00] Wilfred J. Legato, *A Weakest Precondition Model for Assembly Language Programs*, unpublished manuscript, June 19, 2000.

[Moore01] J Strother Moore, Rewriting for Symbolic Execution of State Machine Models, LNCS 2102, **Computer Aided Verification, 13th International Conference, CAV 2001**, Paris, France, July 18-22, 2001, Proceeding

[Wilding97] Matthew Wilding, *Robust Computer System Proofs in PVS*, In **LFM97: Fourth NASA Langley Formal Methods Workshop**, C. Michael Holloway and Kelly J. Hayhurst, eds. NASA Conference Publication no. 3356, 1997

[Wilding01a] Matthew Wilding, David Greve, and David Hardin, *Efficient Simulation of Formal Processor Models*, **Formal Methods in System Design**, 18(3), Kluwer Academic Publishers, May 2001.