vFaat: von Neumann Formal Analysis and Annotation Tool

David Greve and Matthew Wilding
Rockwell Collins Advanced Technology Center
February 12, 2003

Abstract

Formal verification can be used to establish important properties of critical systems. However, applying formal methods to a low-level implementation of a complex system is a daunting challenge, in part because extracting abstract functionality from a specific implementation is tedious. Automating such efforts by placing them under computer control helps free the user to focus on the essence of the verification problem.

vFaat is a tool suite to assist in the formal verification of imperative code executing on von Neumann computing architectures. Building on our experience developing proofs about high-assurance microcoded processors, this work codifies several *ad hoc* techniques to simplify the process of reasoning about software-based systems.

1. The Promise and Challenge of Code Proofs

1.1 Formal Methods and the Code Verification Challenge

Formal methods allow precise descriptions of systems and requirements and enable them to be related in a mathematically meaningful way. Formal verification can demonstrate that, under all conditions of interest, a particular design behaves as specified. A formal proof of correctness can account for every condition that the design might experience regardless of the size of the design's state space. Formal methods therefore provide both the high assurance and the vital scaling property that are necessary for verifying complex designs.

A crucial consideration in formal methods work is the level of abstraction that is appropriate for models. The use of high-level, abstract models as the basis for formal methods verification allows for simpler reasoning when the properties of interest can be conveniently addressed at the algorithmic level. An excellent example of reasoning about such an abstract system is the work at Rockwell Collins analyzing mode awareness in flight guidance systems that formally demonstrates that the design of modern flight automation software does not lead to dangerous ambiguities [Butler98, Miller01].

There are, however, several reasons for adopting a more detailed model of computation in some applications of formal methods.

> **To guarantee the model's fidelity** In order for the evaluation of the system to take advantage of the formal methods, it must be certain that the formal model used to support the proofs actually reflects the behavior of the system being scrutinized. When the model is a low-level model based on the actual code it is easier to demonstrate this connection [Greve00c].
>
> **To reason about low-level primitives** Assembly code and compiler directives are commonly used in low-level software implementations. Such constructs are most common in code of high criticality; code such as operating system kernels and math libraries. Unfortunately, the semantics of these primitives are often difficult to express at the source code level, making formal analysis nearly impossible.
>
> **To avoid reasoning about compilers and other software tools** By reasoning directly about low-level machine code or microcode, one bypasses tools such as compilers and linkers that could impact the correctness of an application.[Greve00b].

The disadvantage of reasoning about code directly, rather than at a more abstract level, is complexity. Code proofs involve many implementation details that would be ignored when reasoning about a more abstract model. Some work has been done on techniques for solving this problem. One notable project is the CLI short stack [Bevier89, Wilding93]. A family of implementations – an assembler, a compiler, a hardware design, and two applications – are shown to work together and are proved correct using a theorem prover. Yuan Yu demonstrated proofs of 68020 code, many of which were compiled into machine code from higher-level languges. [Yu92]. Rockwell Collins has used the PVS theorem proving system to reason about code in several projects [Wilding97, Greve98, Miller99].

This paper describes how we will develop tools to incorporate previously developed techniques into an automated tool supporting a code proof process.

1.2 The CAPS Project: A Critical Microcode Verification Approach

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 applied to several sequences of actual microcode [Greve02, Greve00a, Wilding01a]. Three important aspects of this work were model development tools, proof decomposition techniques, and modeling and proof automation.

- ***Formal Model Development Tools*** Rockwell Collins has developed tools and techniques for writing and reasoning about low level implementations. These methods enable the construction of formalized implementations of sufficient detail to execute device production tests. Such low level models provide high confidence in the fidelity of the model

- ***Code Proof Decomposition*** A crucial challenge in code correctness proofs is developing a methodology for breaking the proof down into smaller, more manageable pieces. The three primary proof decomposition techniques exploited in the CAPS program were the separation of algorithm from implementation, the exploitation of code block structure to break code execution into discrete steps, and loop and block simplification.

- ***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. Three forms of automation employed in the Rockwell Collins' CAPS program were the mechanical generation of proofs, sophisticated reasoning libraries, and fundamental theorem prover enhancements.

Automation of the modeling and proof process in these respects was crucial to being able to demonstrate formal microcode verification in the CAPS project. Nearly all of the techniques described here apply to proofs other than 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.

2. vFaat: A Modeling and Code Verification Tool

The vFaat (von Neumann[1] Formal Annotation and Automation Tool) methodology supports the kind of proof management and automation seen in the CAPS work in a way that is theorem prover and domain language independent.

2.1 The Tool Suite

The vFaat tool assists in the formal analysis of code, such as machine code or microcode. The tool mechanizes and manages standard practices employed in the code proof process. The core of the tool suite is processor independent, making it useful for reasoning about object code for a wide variety of microprocessors. It is also theorem prover independent and can be targeted towards a variety of theorem proving environments.

The tool flow has four basic parts – input, annotation, automation, and output. **Input** is the process of extracting useful information from executable object files. **Annotation** is the process of linking both user-provided and machine generated information to the internal data structures. **Automation** is a set of implementation independent analyses that the tool set provides to the end user. **Output** is the process of converting the internal data structures and annotation into theorem prover proof scripts. Each of these stages is described in more detail in the following sections.

2.1.1 Input

Input is the process of parsing an object file, extracting useful information from it, and storing the resulting information in a format usable by the other tools in the tool chain. This front-end software is object file and instruction set specific, but the information resulting from this process is independent of the object file and

[1] A von Neumann machine exhibits a read-execute-write style of execution, a property of both machine code and microcode execution.

instruction set. The input-processing step isolates the tool chain from the specifics of the object code format and instruction set, ultimately maximizing the applicability of the overall system.

2.1.2 Annotation

Annotation enables the user to insert pre-conditions, post-conditions, function definitions, and proof scripts at specific points in the control flow graph. Annotation is the facility that bridges the gap between those tasks that can be automated within the tool suite and those tasks that must be performed by other means. It is also the mechanism through which vFaat's functionality can be extended by other, third-party tools.

Annotations can be generated either mechanically or by hand. Annotations are *persistent* in the sense that they can be preserved from one tool run to the next. Because annotations represent an investment of effort and because they are persistent, annotations must be managed robustly in the face of minor changes to the underlying code. The implementation attempts to associate annotations with tags linked to labels and other debugging information in order to isolate them from small changes in the target code.

The primary target of annotation is the hierarchical control flow graph. In the hierarchical control flow graph larger blocks are defined by composing some number of smaller blocks. This over-arching block structure is a natural vehicle for managing proofs about such composed functions. As observed in the CAPS work described previously, a proof obligation for a function reduces to a number of simpler proof obligations on the functions of which it is composed. Thus, by associating properties with specific blocks, we are able to manage the decomposition of the overall proof process.

2.1.3 Automation

While the annotation facility supports the management of a proof effort, the automation functionality mechanizes appropriate aspects of the verification task. While not every task is amenable to mechanization, there are a wide variety of tasks that can be performed with little or no user guidance. Of course, the more automation that can be applied to the verification process, the more accessible the process becomes to the typical user. Perhaps more importantly, the more mundane, routine tasks that can be offloaded to a computer the more time the verification engineer can spend focused on the heart of the verification problem.

An important feature of the tool architecture is that it allows functionality to be added to the base configuration in a modular fashion, incrementally improving the power and usability of the system without requiring major re-architecting of the base design. We are evaluating the following automated procedures for incorporation into the tool suite.

> **Control flow analysis**: Control flow analysis allows one to derive automatically block hierarchy from a flat control flow graph. Such analysis assists in the proof decomposition process and has been accomplished in various efforts, including the JEM1 symbolic simulation work [Greve98], the CAPS work [Greve02], and in Destiny [Legato00]. Control flow analysis involves identifying basic blocks, finding and encapsulating loops, removing delayed branches, incorporating exceptions and interrupts, and encapsulating subroutines.
>
> **Weakest Precondition Propagation**: The management of pre- and post-conditions is supported by the annotation facility. However, without automated support, the user is still responsible for generating and propagating these conditions through the control flow graph. The integration of vFaat with a weakest precondition propagation tool such as Destiny [Legato00] would allow such conditions to be automatically pushed through the control flow graph.
>
> **Parity Analysis**: Stacks and queues are common data structures in many computing systems. Good system behavior, however, requires that access to such structures be guaranteed not to overrun the allocated memory area. While such analysis can be tedious if performed by hand, the specific, well-defined nature of the accesses performed on such structures allows much of this analysis to be automated.
>
> **Functional Derivation**: Previous work has explored the automatic derivation of block functionality [Greve98]. Integrating this technology with vFaat would allow the system to generate automatically a symbolic representation of the value of each interesting state element in terms of the initial state of the block.

2.1.4 Output

The final output of the tool suite is a proof script that can be loaded into a general-purpose theorem prover. Unlike a compiler whose failure can introduce an error in the resulting executable, every decision made by vFaat is ultimately checked by a theorem prover. This is important because it allows the introduction of sophisticated, third party analysis tools into the tool chain without requiring the user to trust the soundness of such tools.

The vFaat system is a tool for managing and automating specific aspects of the proof process for imperative code. vFaat is not a theorem prover and its functionality is designed to be theorem prover independent. Targeting vFaat to a new theorem prover is a matter of interpreting the common idioms used by the tool into the language and structure of the targeted theorem prover. vFaat automatically generates only function definitions and simple rewrite rules. The function definitions take the form of block functional descriptions, clock functions, and predicates over the state. The proof obligations take the form of simple rewrite rules and include theorems to enable the connection of the sequential block functions and theorems to demonstrate the invariance of certain properties.

Any interesting proofs that take place in the system must be performed by the user or by third party tools that interface with vFaat through the annotation facility. Additionally, while proofs scripts are almost always necessary to guide the theorem prover in the proof process, these scripts must be coded by the user to target a specific theorem prover and managed by the system as necessary via annotations.

3. Summary

Rockwell Collins has a continuing interest in the development and verification of highly secure software-based communication systems. Rockwell Collins has experience in the field of formal verification of computing systems and has constructed prototype versions of various aspects of the vFaat system for microcoded microprocessors targeting both the ACL2 and PVS theorem provers. We believe that vFaat, being processor and theorem prover independent, will target a wide variety of problem domains and help leverage previously developed verification techniques.

Bibliography

[Bevier89] W.R. Bevier, W.A. Hunt, Jr., J S. Moore, and W.D. Young. An approach to systems verification. **Journal of Automated Reasoning**, 5:411--428, 1989

[Butler98] Ricky Butler, Steven Miller, James Potts, and Victor Carreno. *A formal methods approach to the analysis of mode confusion*. In **17th AIAA/IEEE Digital Avionics Systems Conference**, Bellevue, WA, October 1998.

[Greve98] David Greve, *Symbolic Simulation of the JEM1 Microprocessor*, **Proc. FMCAD'98**, Palo Alto, Nov. 1998, Springer Verlag LNCS N1522, pp.321-333

[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)

[Greve00b] David Greve, Matthew Wilding, Mark Bickford, and David Guaspari, Orpheus: A Self-Checking Translation Tool Arrangement for Flight Critical Hardware Development, **Langley Formal Methods 2000 -- LFM00,** Williamsburg, VA, 2000**.**

[Greve00c] David Greve and Matthew Wilding, Executable Formal Models for Validation and Specless Verification, **19th Digital Avionics Systems Conference (DASC)**, Philadelphia, PA, October 2000.

[Greve02] David Greve and Matthew Wilding, Evaluatable, High-Assurance Microprocessors, Proceedings of High Confidence Software and Systems Conference (HCSS), Linthicum, MD, March 2002.

[Kaufmann00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, **Computer-Aided Reasoning: An Approach**, Kluwer Academic Publishers, 2000. (ISBN 0-7923-7744-3)

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

[Miller99] Steve Miller, David Greve, Matthew Wilding, and Mandayam Srivas, *Formal Verification of the AAMP-FV Microcode*, NASA Report NASA/CR-1999-208992, Feb 1999.

[Miller01] Steven P. Miller and Alan C. Tribble, Extending the Four-Variable Model to Bridge the System-Software Gap, Presented at the **20th IEEE / AIAA Digital Avionics Systems Conference (DASC)**, Daytona Beach, FL, October 2001.

[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

[Rushby01] John Rushby, Security requirements specifications: How and what?, Invited paper presented at **Symposium on Requirements Engineering for Information Security (SREIS)**, Indianapolis, IN, March 2001.

[Shankar00] Natarajan Shankar, *Efficiently Executing PVS*, SRI Computer Science Lab report, Nov. 1999.

[Wilding93] Matthew Wilding, A Mechanically Verified Application for a Mechanically Verified Environment, **Computer-Aided Verification -- CAV '93**, Springer-Verlag Lecture Notes in Computer Science volume 697, 1993.

[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.

[Yu92] Yuan Yu, *Automated Proofs of Object Code for a Widely Used Microprocessor*, Ph.D. Thesis, The University of Texas at Austin, 1992