Symbolic Simulation of the JEM1 Microprocessor

David A. Greve

Rockwell Collins Advanced Technology Center
Cedar Rapids, IA 52498 USA
dagreve@collins.rockwell.com

Abstract. Symbolic simulation is the simulation of the execution of a computer system on an incompletely defined, or symbolic, state. This process results in a set of expressions that define the final machine state symbolically in terms of the initial machine state. We describe our use of symbolic simulation in conjunction with the development of the JEM1[1], the world's first Java[2] processor. We demonstrate that symbolic simulation can be used to detect microcode design errors and that it can be integrated into our current design process.

1 Introduction

Traditional microcode verification techniques, which include extensive testing and demanding design reviews, have historically provided us with reasonable levels of assurance concerning the correctness of complex microcoded processors. However, the high cost of failure associated with devices used in ultra-critical applications or those undergoing mass production demands that the verification techniques employed in the design of that device provide extremely high confidence of correct design functionality, even in the face of ever increasing design complexity and reduced time to market.

Formal verification provides a high level of confidence in the functionality of a design by establishing that a formal model of an implementation satisfies a given formal specification. Figure 1 presents the standard formal verification commuting diagram. Typically, the formal verification of a microcoded microprocessor involves proving that the sequence of microinstructions f1,f2,...,fn result in a state change at the microarchitecture level that corresponds, via some abstraction function, to the state change resulting from the application of the machine instruction, F, at the macroarchitecture, or programmer's, level.

Although formal verification provides the highest degree of certainty that an implementation meets a given specification, our experience with programs employing formal verification is that it is time consuming, expensive, and not

[1] JEM and JEM1 are trademarks of Rockwell

[2] Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

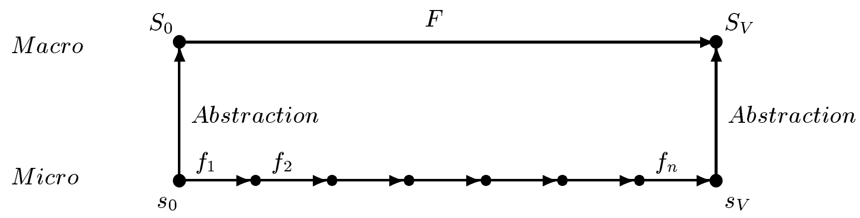

Fig. 1. Formal Verification Commuting Diagram

sufficiently mature to incorporate directly into our design process. We are exploring alternatives to formal verification that can provide high degrees of assurance: symbolic simulation is one such alternative.

Symbolic simulation is the simulation of the execution of a computer system on an incompletely defined, or symbolic, state. This process results in a set of expressions that define the final machine state symbolically in terms of the initial machine state. One uses an automated reasoning tool to derive a symbolic expression for the behavior of a computer system from a formal model of that system, a process corresponding to the bottom sequence of arrows in Figure 1. However, to establish the ultimate correctness of the derived behavior, corresponding to the upper segments in Figure 1, symbolic simulation relies on visual inspections of the results rather than mathematical proof.

Symbolic simulation is therefore a semi-formal technique which attempts to capitalize on the advantages of formal verification. By eliminating the need for formalizing and maintaining a high level specification as well as the need to derive formal proofs of correctness we reduce time and cost. By retaining the formal model of the implementation and using automated reasoning tools to derive expected behavior we retain much of the value of formal verification.

In this paper we review the goals and objectives of a recently completed symbolic simulation project, illustrate some of the techniques used in implementing the program, and discuss the outcome of the program. We also present several suggestions and observations that should be considered by future programs employing symbolic simulation.

2 Background

Our group has experimented with the application of formal methods to the verification of microcoded microprocessors [8, 7, 12]. The primary goal of these programs was to evaluate the viability of formal methods in industrial settings. As previously mentioned, although formal verification can be used to uncover design errors, our experience has indicated that it has a serious drawback: the methodology is still too immature to apply in an industrial setting.

We discovered in applying formal verification that many of the errors that we ultimately detected in our formal models were revealed during the symbolic simulation of the microcode, rather than during equivalence checking with an abstract specification. Individuals acquainted with the expected operation of a sequence of microcode can often identify errors in symbolic expressions easily because they manifest themselves as unusual looking results. The implication of this discovery is that the mere symbolic simulation of microcode is nearly as valuable as full formal verification in many situations. An early study of symbolic simulation can be found in [1]. The use of formal methods in the absence of an abstract specification was also explored in [5].

We also observed that, to a large extent, this type of symbolic analysis is amenable to automation. In the case of simple sequential microcode with the appropriate infrastructure in place, one need only indicate to the automated reasoning system the number of microcycles required to execute the sequence. Armed with this information the reasoning system can automatically expand definitions and rewrite terms to compute the symbolic result of that sequence of code. This simple technique does not necessarily apply to microcode loops, but it is a relatively simple matter to identify such sequences and deal with them on a case by case basis.

We concluded that symbolic simulation is the aspect of formal verification that is currently the most viable in our design environment. It retains much of the value of formal verification by retaining the formal model of the implementation and using automated reasoning tools to derive expected behavior. It also provides a simple, largely automatable methodology that makes it applicable in an industrial setting. Finally, by eliminating the need for developing and maintaining a high level specification and its associated, often fragile, formal proofs of correctness, symbolic simulation minimizes the time and cost of implementing the program.

This paper describes how symbolic simulation was applied in development of the JEM1, the world's first Java processor. We begin in Section 3 with a brief overview of the microarchitecture of the JEM1. Section 4 describes the elements of the symbolic simulator itself and Section 5 provides the results of the JEM1 symbolic simulator effort.

3 JEM1 Microarchitecture Overview

The JEM1 is the world's first direct execution Java processor [13, 4], a processor whose instruction set is a superset of the instructions specified for the Java Virtual Machine (JVM) [6, 2]. For a machine supporting an Instruction Set Architecture (ISA) as sophisticated as that of the JVM, the hardware implementation of the JEM1 is surprisingly simple. The JEM1 is a classic microprogrammed machine, including a control store ROM, microsequencer, and datapath, with nearly every processor operation controlled by microcode.

A crucial element of the microarchitecture is the control store ROM which, in a microprogrammed processor, contains the microcode that controls the machine. Each word in the control store ROM represents a single microinstruction which, in turn, corresponds to one line of microcode. Each microinstruction is logically partitioned into several fields, with each field responsible for controlling a specific portion of the machine. Because the JEM1 implements nearly all of the JVM instructions directly in microcode, the control store ROM is large and the resulting microcode is quite complex.

The microsequencer circuit is responsible for generating the microaddresses that are used to access the microcode ROM. The behavior of this circuit is largely under microcode control and allows the microprogrammer to sequence to the next sequential microcode location, to conditionally or unconditionally jump to a specific location, or to perform a single level of call and return. It also provides the ability to parse the next opcode and vector to its associated microcode entry point.

The datapath provides a multiport register file, a 32 bit ALU and barrel shifter, as well as status registers and shift linkages. This basic hardware is sufficient to support microcode implementations of all of the JVM defined arithmetic instructions over 32 and 64 bit integers and floating point numbers.

4 The JEM1 Symbolic Simulator

The JEM1 symbolic simulator is conceptually composed of several layers. The base of the simulator is the formal reasoning system which in our case is the PVS reasoning system. PVS is used to construct a formal specification of the JEM1 and its associated microcode. This model is ultimately controlled by a set of supporting software which assists in the automation of the simulation process. Finally, at the top level is the simulation environment itself. In the subsequent sections we describe each of these components in more detail.

4.1 PVS

PVS is the automatic reasoning system upon which we built the JEM1 symbolic simulator. Developed at SRI International's Computer Science Laboratory, PVS (Prototype Verification System) is an automated reasoning system for "specifying and verifying digital systems" [11, 9, 10]. The system consists of a specification language, a parser, a typechecker, and an interactive proof checker. It supports a specification language that is based on a simply typed higher-order logic, and provides a large number of prover commands that allow machine-checked reasoning about expressions in the logic. The primitive proof steps involve, among other things, the use of arithmetic and equality decision procedures, automatic rewriting, and BDD-based Boolean simplification. There is also support for the automation of reasoning in PVS via a facility for constructing new proof commands.

For the purposes of modeling simple digital logic, PVS provides the built-in Boolean type, bool, and most of the primitive Boolean operators including NOT, AND, and OR, IF-THEN-ELSE, CASES and equality. It is also possible to develop a more abstract model of digital logic using the enumerated type system provided by PVS. PVS provides direct support for reasoning about enumerated types and allows their use in CASES and equality expressions.

PVS allows the overloading of operators, including the built-in operators, to work with arbitrary types. This overloading capability allows us to use symbols that closely resemble those normally found in the digital logic domain when describing hardware functionality. PVS does not, however, provide a built-in type for modeling groups of bits, herein referred to as bitvectors. For this reason, a great deal of effort was required to develop and evaluate various techniques for representing and manipulating such constructs.

4.2 Formalization of the JEM1

Using the logic of PVS to describe the microarchitecture of the JEM1 is the first step. The microarchitecture specification is a formal description of the hardware which implements the JEM1 processor. Each major component of the microarchitecture, such as the ALU or the register file, is described in one or more PVS theories which defines its internal state, inputs, and outputs. These specifications, together with a variety of "glue" theories describing the data and control paths between them, define the microarchitecture over which the microcode executes.

The microarchitecture specification describes the processor from the perspective of a microcode programmer and abstracts away some of the details of the actual hardware implementation. Time, for example, is modeled in the microarchitecture using the natural numbers, where one unit of time corresponds to one *microcycle*, the time required for execution of a single microinstruction. Although a memory transaction may take an arbitrary amount of real time to complete, from the perspective of the microcoder each memory access takes a single microcycle. By abstracting away from the physical clock, it is possible to provide a more concise definition of the microarchitecture and automate much of the simulation process.

The microcode ROM is modeled as an uninterpreted function that accepts as input a microaddress and produces as output a microinstruction. Defining a line of microcode in the ROM involves the introduction of an axiom that states that the value of the ROM function evaluated at the microaddress of interest is the desired microinstruction. The process of constructing these axioms has been automated and uses the same files used to program the ROM in the actual device. We have chosen to use axioms rather than deriving similar results from a functional definition to minimize the time required for PVS to typecheck the theories associated with a particular microcode segment.

4.3 Supporting Software

Several aspects of symbolic simulation benefit from informal software analysis. For the JEM1 symbolic simulation program we crafted special purpose software to perform microcode translation, loop elimination and symbolic output reformulation.

As previously mentioned, the microcode translation software converts the output of the microassembler, including field names and enumeration values, into the format outlined by the formal PVS microcode representation used at the microarchitecture level. Figure 2 presents output of the microcode translation program. The **Label_IAND** lemma states that the value of the symbolic label **IAND** is microaddress 1918. The **uAX_IAND** axiom then defines the microinstruction located at location 1918 of the microcode ROM.

```
uCode_IAND : THEORY

BEGIN

  IMPORTING uCode_Labels,uCode_Definition

  Label_IAND : LEMMA IAND = 1918

  uAX_IAND : AXIOM uROM(1918) = (#
BS                              := FETCH_lt_1,
BS32                            := BS32eq11,
NM                              := MAP,
MODIFY                          := udef_MODIFY_31,
OVR_LK_CTRL                     := LOCK,
CNTR_CTRL                       := LD_CNTR,
SPARE_CTRL                      := udef_SPARE_CTRL_7,
TS                              := STATUS,
STATUS_REG                      := STgetsTFF2,
NIBL                            := NIBL_F,
TAU_CTRL                        := TAU_CTL,
CT                              := PCplus1,
PP                              := POP1,
RS                              := RisA,
SS                              := SisB,
FN                              := RandS,
FN3                             := FN3is1,
DN                              := BgetsFN,
CARRY_SELECT                    := CIisZ,
SIGN_REG                        := SIGNgetsSIGN,
IM_DATA                         := ZEX_IMx4,
AA                              := AisV,
BA                              := BisVminus1,
MIN_STACK                       := SVgt1,
MAX_STACK                       := NO_MAX,
JA                              := 4095,
uC                              := -1,
        { ... }
  #);

END uCode_IAND
```

Fig. 2. Partial PVS Specification of IAND Microinstruction

The process of loop elimination begins with the development of an informal software model of the JEM1 microsequencer. As mentioned in Section 3, the microsequencer for the JEM1 is quite simple and so it is easy to model its behavior in software. This informal model of the microsequencer enables us to perform software analysis of the control flow graph resulting from the evaluation of the JEM1 microcode using that model. This control flow analysis provides a convenient means of identifying and encapsulating microcode loops. It is also possible, using the same control flow analysis, to locate shared code segments. Once the software has identified such constructs, it partitions the microcode into segments containing only sequential, non-looping microcode sequences and generates a clock function for each of those segments. The clock function is a PVS function that calculates how many microcycles are required to execute the given microcode sequence based on the current state of the machine. These microcode segments can then be presented to PVS for symbolic simulation. Because the segments are sequential and contain no loops, using the supplied clock function we are able to direct PVS to automatically perform the simulation of the entire segment and produce the desired symbolic results.

Once the simulation is complete, PVS dumps symbolic results into a file. These results are run through a final conversion tool that reformulates the symbolic results into a representation that is easier to read and which can be fed back into PVS.

4.4 Simulation Environment

The symbolic simulation itself is performed completely within the confines of PVS. The top level symbolic simulation lemma for any given sequence of microcode claims that the state of the processor following the execution of that sequence of microcode is the same as it was initially. This conjecture is not expected to be a theorem; rather, the process of deriving a counterexample forms the basis of symbolic simulation.

A generic proof script is executed on the top level lemma which causes the final state of the processor to be derived via symbolic evaluation of the sequence of microcode under consideration. The evaluation of the microcode is performed with the theorem prover via expansion of function definitions and the rewriting and simplification of terms. The final steps in the proof script replace the equality over the state with a conjunction asserting that each element of the processor state has remained unchanged. When PVS fails to prove this assertion, what remains is a simplified conjunction containing only those assertions concerning the state elements that have in fact changed. PVS then dumps this unproven conjunction into a file. This unproven conjunction is the raw result of the symbolic simulation. A portion of the file resulting from the execution of the IAND microcode sequence is shown in Figure 3.

The file containing the raw symbolic simulation results is then run through the reformatting tool to generate a reformulation of the top level symbolic simulation theory. This reformulation results in a top level lemma that claims that the state of the machine following the execution of the microcode is equal to the symbolic simulation results just obtained. By incorporating this change and re-running the symbolic simulation proof script, it is possible to perform simple formal regression testing. In such a case, one expects PVS processing of the proof script to succeed. Figure 4 illustrates the reformulation of the results for the portion of the IAND microcode simulation presented in Figure 3. It is these

```
uSim_IAND-uSimulation.sequents

uSimulation :

  |-------
{1} ((pc |= PC(T0) + 1) = (pc |= PC(T0)))
  & ((vector |= V(T0) - 1) = (vector |= V(T0)))
  & ((skmt |= (V(T0) = 0)) = (skmt |= (1 + V(T0) = 0)))
  & ((Vm0 |= stack(1) AND stack(0)) = (Vm0 |= stack(1)))
  & ((carry |= unspecified_CARRY(T0) ^ (32))
     = (carry |= CARRY(T0)))
  & { ... }
```

Fig. 3. Partial sequent file for the IAND microcode sequence

reformulated results that we provide during the microcode design reviews to assist in establishing the correctness of the code.

We want to emphasize that the entire symbolic simulation process, including the microcode conversion, the running of the simulation, and the reformulation of the symbolic results, is completely automated. The only user interaction required is the initiation of each of these tasks.

5 Program Results

By the end of the symbolic simulation project, the JEM1 control store ROM contained 1689 lines of microcode. The process of eliminating loops and identifying common subroutines in this microcode ultimately produced 521 unique microcode segments, or approximately 3 lines of microcode per segment.

Each of these microcode segments results in five PVS files: one defining the microcode, one defining the clock function, one stating the clock function in terms of rewrites rules, a next state theory, and a top level simulation file. Each of these files also has an associated PVS proof script. The entire JEM1 symbolic simulator specification, including bitvector libraries, requires 5.7 megabytes of disk space.

The JEM1 symbolic simulation effort lasted for approximately 6 months and was manned at a level of approximately one half an engineer during that time. In the course of this program, we were able to perform symbolic simulation for all of the 1689 lines of microcode in the control store ROM. Symbolic simulation results, however, were available for only 3 of the early microcode walkthroughs, constituting coverage of 62 of the 518 microcode segments.

5.1 Issues

This program served to highlight three specific weaknesses in our symbolic simulation techniques: slow symbolic execution; the need for model validation; and the inability to represent complex symbolic results clearly.

```
Next_IAND[(IMPORTING uState_Definition) T0: time]: THEORY

BEGIN

  st: VAR micro_state

  i : VAR uState_Record

  IMPORTING uCode_Labels

  IMPORTING uSim_AbstractDefs[T0]

% Branch 0 :

  F_0(st)(i) : uState_Element[uState_Record, i] =
    CASES (i) OF
        pc       : pc       |= PC(T0) + 1,
        vector   : vector   |= V(T0) - 1,
        skmt     : skmt     |= (V(T0) = 0),
        Vm0      : Vm0      |= stack(1) AND stack(0),
        carry    : carry    |= unspecified_CARRY(T0) ^ (32),

        { ... }

        ELSE       st(i)
    ENDCASES;

  NextSt(st)(i) : uState_Element[uState_Record, i] = F_0(st)(i);

END Next_IAND
```

Fig. 4. Partial reformulated results for the IAND microcode sequence

Simulation Speed The slow symbolic simulation speed of PVS was perhaps the greatest detriment to this program. The fact that it took several minutes to execute a single line of microcode impacted the rate at which bitvector libraries could be evaluated, the turnaround time involved in finding and fixing model and microcode errors, and ultimately our ability to produce symbolic results in time for upcoming walkthroughs.

The CPU time required to perform the symbolic simulation for all 1689 lines of microcode was nearly 17 days, not including typechecking. This averages out to approximately 14 minutes per line of microcode. In general, it took around 6 minutes to execute a microcode segment containing only a single line of microcode. However, the time required to execute segments containing more than one line of microcode was apparently super-linear in the number of lines of code in the sequence. For example, from Table 1 we can see that approximately 20% of the total simulation time, corresponding to the top four entries in the table, was spend on just 7 particularly long and complex microcode segments. All times are for PVS 2.0 Alpha Plus (patch level 2.394) running on a Sparc 20 with 96 MB of main memory.

Microcode Segment(s)	*time (sec)*
BOOT_RET_4	18845
F2I	11752
DDIV+F2L+D2I+D2L	72603
BIST_CONT	190364
others	1143780
Total	1437344

Table 1. Simulation Times for Selected Microcode Segments

Future symbolic simulation efforts will require techniques for improving the throughput of the formal reasoning system used to perform the actual symbolic simulation.

Model Validation Another issue which exacerbated the problem of simulator performance involved errors in the formal model at the time of the design reviews. These errors existed because the processor formalization itself had never undergone any other form of model validation. An interesting impact of the decision to not validate the processor model up front was that it moved some of the cost of establishing model correctness from early in the program, during model development, to late in the program, during microcode inspections. The symbolic simulations ultimately exercised many of these errors which subsequently became apparent either during the simulation or the subsequent microcode walkthrough. Model errors required the regeneration and revalidation of simulation results. This task had to be performed in real time between design reviews. Unfortunately, due to the speed of the symbolic simulator, this was often not possible.

Efforts are already underway to explore the use of executable formal models as a part of the standard design process [3]. By using a single model as both a conventional microcode simulator for use in microcode development and as a basis for symbolic simulation, it is believed that the model validation issue can be resolved.

Result Presentation One daunting challenge of symbolic simulation is the concise representation of symbolic results. In order to maintain the high value of formal verification in the absence of formal equivalence checking, it is essential that the microarchitects be capable of reviewing the symbolic results with enough understanding to detect the sometimes subtle nuances which distinguish modeled behavior from desired behavior. In cases where the microarchitect cannot interpret the symbolic results, much of the value of symbolic simulation is lost.

Symbolic simulation results can often represent intermediate steps in a complicated numerical computation. For this reason, it is essential to have a bitvector library that is capable of reducing common bitvector operations, including arithmetic, logical operations, bit extraction, bit concatenation, and arbitrary combinations thereof, into the simplest possible form. Note that this requirement is not the same as requiring bitvector decision procedures. The objective of the bitvector library is to simplify the presentation of a bitvector operation, not necessarily to establish the equality of two bitvector expressions. The bitvector libraries used in this program were adapted from those used in our previous formal verification and were therefore relatively complete. However, there is still substantial room for improvement in this area.

The more difficult cases involved the longer, more complex sequences of microcode, especially those that performed multiple memory updates and dereferences. Such sequences resulted in symbolic expressions that were nearly impossible to read. This problem was addressed to some extent by providing the simulator with local knowledge of the memory mapped data structures being manipulated. Unfortunately, this is difficult to automate and doesn't work well in every case. Deriving a truly palatable representation for symbolic results in general is still, to a large extent, an open issue.

It should be noted that, even in the case where the symbolic results are difficult to read, they can still provide value if used as a regression test suite. By including symbolic simulation results as a part of this test suite, it is possible to load previous symbolic results back into the automated reasoning system and provide formal assurance that the behavior of the microcode has remained unchanged. As mentioned in Section 4.4, our current tool set already supports this approach.

5.2 Advances

Although the symbolic simulation program suffered from some significant setbacks, we believe that it was successful in demonstrating that it is possible to use symbolic simulation to find microcode errors and that the process can be automated and used in an industrial setting. These two aspects are discussed in greater detail in the following sections.

Incremental Improvement Symbolic simulation provides an intermediate step between our current verification approach, involving design reviews and functional testing, and full formal verification. The primary goal of symbolic simulation is to provide an incremental improvement over the current verification methodology. We believe that by demonstrating that microcode errors can be detected through the inspection of symbolic results, we have met this goal. Some examples of the problems identified in the course of the symbolic simulation program were:

An extraneous memory transaction. The incorrect specification of a particular microoperation resulted in a line of microcode that unexpectedly generated a write transaction to memory. The incorrect memory transaction was obvious in the symbolic results because no memory transactions were expected to take place during the microcode sequence.

Incorrectly specified shift value. A line of microcode erroneously employed a shift value of 15, rather than the desired value of 16. The fact that the symbolic result would not simplify and given that the task at hand was to create a 32-bit bit mask, the shift value of 15 appeared unusual.

Unintended alteration of a register value. A line of microcode employed a microoperation which, in conjunction with other microoperations in the microinstruction, resulted in the unintended side effect of zeroing a specific state register. This error was actually detected as part of a routine microcode testing procedure. The result, however, was verified by the symbolic simulation.

Use of an unspecified value. An error in the formal processor model resulted in a microcode branch based upon the value of an operation which, in the model, was unspecified. This particular error was easily identified because the term "unspecified" appeared in the formulation of the branch condition in the symbolic result.

A microassembler inconsistency. The tools employed to automate the formalization of the microcode revealed an inconsistency in the software used to assemble the microcode.

It is likely that all of these errors would have been discovered by traditional verification techniques, although some sooner than others. However, they are presented here to underscore the fact that many microcode errors are trivial to identify symbolically simply because they manifest themselves as unusual looking results.

It should be noted that, in addition to providing the basis for detecting errors via informal inspections, symbolic results also can act as the first step in the larger process of full formal verification. A program like the one we employed with the JEM1 is attractive because it allows one to partition the formal verification problem into two manageable portions, symbolic simulation and then formal equivalence checking with an abstract specification, with value being added in each step.

Automation As previously mentioned, one of the advantages of symbolic simulation over formal verification is that, because symbolic simulation is amenable to automation, it can be more easily integrated into the current design environment. We feel that the JEM1 symbolic simulation program was a significant demonstration of the ability to automate this process. By employing software tools to convert and analyze the microcode under consideration, the entire process, from translation to simulation to formulating the results, has been automated. In this particular program our ability to automate the process far outstripped our ability to produce and consume the symbolic results.

6 Conclusion

Symbolic simulation is an incremental improvement over the current verification process and a possible step towards inserting formal verification into the traditional design cycle. We have demonstrated this capability in the design cycle of the world's first Java processor, the JEM1. Although some deficiencies in our system hindered our work, we believe that we have demonstrated the validity of this approach as well as its potential. SRI is currently at work to improve the performance of PVS for applications such as ours and our future work in this area will study the impact of faster theorem provers, different formal representations, and extensions to the current analysis tool suite to provide even greater levels of automation.

References

1. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. Technical report, Stanford Research Institute, Menlo Park, CA, 1975. CSL-20.
2. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, Reading, Massachusetts, 1996.
3. David Greve, Matthew Wilding, and David Hardin. Efficient simulation using a simple formal processor model. Technical report, Rockwell Collins Advanced Technology Center, April 1998. (available at http://home.plutonium.net/ hokie/docs/efm.ps).
4. David A. Greve and Matthew M. Wilding. Stack-based Java a back-to-future step. *Electronic Engineering Times*, page 92, January 12, 1998.
5. Robert B. Jones, Carl-Johan H. Seger, and David L. Dill. Self-consistency checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
6. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading, Massachusetts, 1996.
7. Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.
8. Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.
9. S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
10. S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
11. N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
12. Matthew M. Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA Conference Publication no. 3356, 1997. (http://atb-www.larc.nasa.gov/Lfm97/).

13. Alexander Wolfe. First Java-specific MPU rolls. *Electronic Engineering Times*, page 1, September 22, 1997.