Robust Computer System Proofs in PVS

Matthew M. Wilding

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

Abstract

Practical formal verification of complex computer systems requires proof robustness and efficiency to protect against inevitable mistakes and system specification and design changes. PVS is a theorem-proving system based on higher-order logic with which we demonstrate the kind of robust code proofs needed for verification of realistic-sized computing systems.

1 Introduction

Computer system correctness can be difficult to establish. Formal proofs about formal models of computer systems have the potential to improve the reliability of computer system designs, but they have several drawbacks. Formal proofs about computer systems are often very complex and hard to get right, and the social process that is usually counted on to certify mathematical proofs is ineffective because particular computer system designs are often proprietary and in any case not of general interest. Mechanical theorem provers can help overcome both of these problems with formal proof: proofs generated with computer programs can be easier to produce and more reliable.

PVS is a verification system for "specifying and verifying digital systems" [12, 13, 16]. 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. There is support for automating reasoning in PVS, namely a simple rewriting system and a facility for constructing new proof commands, although the emphasis in PVS is on building clear specifications and supporting user proof with domain-specific decision procedures.

The Rockwell AAMP5 and AAMP-FV are processor designs with microcoded instruction sets. Partial microcode correctness of these processors has been established using PVS [9, 10]. The hardware that executes microcode has been formalized in the PVS logic, and proofs that the microcode correctly implements some of the processor instruction sets have been constructed. While the application of PVS to realistic-sized processors in the AAMP5 and AAMP-FV projects led to a partial verification of their microcode, the experience of building these proofs led the developers to the pragmatic realization that practical computer systems proofs must be robust [9]. That is, computer system proofs must be able to demonstrate correctness with minimal human assistance despite modest system or specification changes.

Mistakes in proof development and changes to system design and specification are inevitable for realistic-sized verifications. For example, during the AAMP-FV verification effort a change was made in the formal model related to memory address decoding [9]. This change caused every previously-constructed instruction correctness proof to fail even though the change had little to do with the substance of most of the proofs. Large programming projects use software engineering techniques to make software robust despite inevitable changes. So too must large machine-checked proof projects use techniques to develop robust proofs.

Various projects besides the AAMP5 and AAMP-FV verifications have established computer system correctness using mechanical proof. A Piton [11] program that plays the puzzle-game Nim is proved to play optimally [17]. Compiled routines from the C string library and elsewhere targeted to the Motorola 68020 are proved to meet their specifications [5]. Microcode for the Motorola CAP processor is proved to implement several algorithms useful for digital signal processing [6]. Others verifications involve a stack of verified systems [2], an operating system kernel [1], code for simple real-time systems [18], and floating-point microcode [6, 15]. Each of these projects employed the theorem proving system Nqthm [3] or its successor ACL2 [8].

The logics supported by Nqthm and ACL2 are weaker than that supported by PVS: they do not conveniently support higher-order functions and quantification. The style of proof encouraged by the theorem proving system is also quite different: Nqthm and ACL2 provide several automatic proof techniques that are programmed by the user by proving theorems and adding them to the theorem prover database. A considerable amount of strategic planning is required to coopt the Nqthm and ACL2 proof heuristics to prove interesting theorems. However, the style of proof of these efforts has an important benefit: proof robustness. Since the proofs are "automatic" – at least in the shallow sense that the same proof heuristics are applied for every proof albeit with different rules databases – even dramatic changes in the system or the specification typically do not render old proofs obsolete. For example, when the verified processor FM8501 was redesigned to increase its wordsize the Nqthm proof of the modified processor correctness theorem worked with minimal human assistance [11]. Theorem provers based on first-order quantifier-free logic have been successful on larger system correctness problems in part because their mostly-automatic approach to guiding the theorem prover.

This paper explores how to use PVS to reason about computer systems in a robust style. We do this by adapting the computational specification style of the Nqthm/ACL2 verifications and by developing a specification and proof methodology that allows relatively automatic PVS proofs about code execution. The proofs employ some of the techniques used in the Nqthm/ACL2 proofs plus some PVS-specific techniques. The use of interpreters to define languages and the automation to improve proof resilience transcend particular theorem provers. However, this approach does not require that we forego the use of the full PVS language and prover in other proofs: we can use our theorems about code execution to prove whatever we wish using the full PVS logic and prover.

In the next section we present a formalization of a simple computing system in order to aid the exposition of this paper. Section 2 outlines our approach for proving code in PVS, using a simple computing system to illustrate our technique. Section 3 gives an example of how the full PVS language can be used for specification in concert with our robust proofs. Section 4 presents some brief conclusions.

2 Reasoning about Program Execution

We describe in this section how to specify and reason about code in a robust way. We introduce a simple machine formalized in the PVS logic with which we illustrate our approach. Two example programs for this machine are presented for which we construct code execution correctness statements. The proofs of these correctness statements are very simple owing to the creation of some simple reasoning support we have built into PVS and some simple conventions we follow in the expression of code correctness. The style of proof is similar in some respects to other verification projects, particularly [5, 11, 17]. These proofs are less sensitive to changes and therefore more robust.

2.1 A Simple Machine Interpreter

In order to make the ideas of this paper concrete we introduce a PVS computing machine formalization that supports examples in later sections. We present **sm**, a slightly modified version of John Rushby's formalization of Bob Boyer's and J Moore's simple machine-level language [4, 14].

An **sm** state is composed of five elements: a program counter, a stack containing subroutine call return addresses, a data memory that maps natural number addresses to natural number values, a flag whose boolean value indicates whether the processor is halted, and a program memory that maps natural number addresses to instructions. We fix both instruction and data memory size at 100 elements which limits the valid addresses for the memories to values less than 100, and represent an **sm** instruction as a record containing one of 13 opcodes and two addresses. The instructions are described informally in Figure 2.1.

The PVS function `step` defines precisely the effect of executing the instruction pointed to by the pc, thereby providing a formal version of the instruction descriptions of Figure 2.1 with which we can reason

move a b store value at location b in location a
movei a n move value n in location a
movewind a b store value at location b in location stored at location a
moverind a b store value at the location stored at location b in location a
add a b store sum of values at locations a and b in location a
sub a b store in location a the greater of 0 and the difference of a and b
incr a increment value at location a
decr a decrement value at location a
jump n store value n in pc
jumpz a n store value n in pc if value at location a is 0.
call n store (incremented) pc on the stack and store value n in pc
ret store a value popped from the stack in pc
halt set the halt flag

Figure 1: The **sm** Instructions

about programs. We define a function `sm` that returns the state resulting from running `n` instructions starting in state `s`.

```
sm(s: state, n: nat): RECURSIVE state =
  IF n = 0 THEN s ELSE sm(step(s), n - 1) ENDIF
 MEASURE n
```

This computing machine is considerably simpler than formalizations of actual machines but it provides enough complexity for sufficiently interesting examples. The specification style of **sm** is similar to many Nqthm and ACL2 efforts, but one difference that exists between **sm** and those other models illustrates an important difference between the styles encouraged by Nqthm/ACL2 and PVS. While **sm** memory is represented by a function, memory in the Nqthm and ACL2 code proof interpreters is represented by a particular datastructure implementation. For example, memory in the formal model of the FM9001 is represented by a binary tree of memory elements [7]. This style difference stems from a difference in proof system functionality. Nqthm/ACL2 provides execution of definitions and encourages concrete, efficient models. (An ACL2 interpreter for a commercial processor executes microcode programs faster than the executable processor model being used for microcode development [6].) PVS cannot conveniently simulate machine execution but provides higher-order logic and encourages specification unburdened by irrelevant detail.

```
address   code
   0      move 2 0
   1      move 3 0
   2      move 4 1
   3      sub 4 2
   4      jumpz 4 12
   5      incr 2
   6      moverind 4 2
   7      moverind 5 3
   8      sub 5 4
   9      jumpz 5 2
  10      move 3 2
  11      jump 2
  12      ret
```

Figure 2: **sm** min subroutine

2.2 An sm Program and Specification

We mainly use two features of PVS to prove program properties: automatic rewrite rules and strategies. We use example **sm** code to illustrate our approach to code correctness proofs. Figure 2 presents a "min" program that returns in register 3 the location of a least element of the array whose bounds are contained in registers 0 and 1.

We specify the behavior of this program using a PVS function that calculates the result using the same algorithm as the **sm** program.

```
least(max, cur, low, mem): RECURSIVE nat =
 IF (cur < max)
  THEN least(max,cur+1,
             IF mem(cur+1)<mem(low)
              THEN cur+1 ELSE low ENDIF,mem)
  ELSE low ENDIF
 MEASURE max(0,max-cur)
```

For convenience and readability we define functions to return the value of registers, so for example `R0(s)` for **sm** state `s` returns the value of `mem(s)(0)`. Also for convenience we define functions `write`, `goto`, and `update_stk` which update respectively the memory, program counter, and call stack of an **sm** state.

We write a function that calculates the number of instructions that are processed during execution of the subroutine. For the min subroutine example, this function is `min_clock`. The structure of the "clock" functions parallels the structure of the blocks of code in the program and is used to guide the proof. The function `clock_plus` is equivalent to natural number plus and is used in clock function definitions to keep the PVS prover from simplifying the expressions. The constant `N` is the size of **sm** data memory.

```
min_loop_once_clock(s):nat =
 if R2(s)+1<N AND R3(s)<N AND
    mem(s)(R2(s)+1)<mem(s)(R3(s))
  THEN 10 ELSE 8 ENDIF

min_loop_clock(s): RECURSIVE nat =
 if pc(s) = 2 AND defs(s) = program
  AND NOT halted(s)
  AND 5<R0(s) AND R0(s)<=R3(s) AND R3(s)<=R2(s)
  AND R2(s)<R1(s) AND R1(s)<N
 THEN
  clock_plus (min_loop_once_clock(s),
   min_loop_clock(sm(s,min_loop_once_clock(s))))
 ELSE 3 ENDIF
 MEASURE max(0,R1(s)-R2(s))

min_clock(s): nat =
 clock_plus (3,
  clock_plus (min_loop_clock(sm(s,3)), 1))
```

We have chosen a specification style that relies on specifying the complete result of a computation because it simplifies the task of automating proofs involving code. A drawback of this philosophy is that unimportant but hard-to-describe elements must be specified too. We specify the value of these irrelevant state elements using functions defined with the interpreter function in a manner that allows us to specify conveniently the entire state resulting from a computation.

An example of this kind of state element occurs in the min subroutine loop. After each iteration register 5 contains the difference between the least element so far encountered and the current value being checked. Although we could of course specify the final value of the loop for register 5, we would prefer to ignore it since the ultimate value of this temporary register is unimportant. We define a function that calculates the final value of the register using the interpreter:

```
min_loop_unspecified_R5(s):nat =
 R5(sm(s,min_loop_clock(s)))

min_correct_unspecified_R5(s):nat =
 min_loop_unspecified_R5(sm(s,3))
```

Using this function to specify the final value of register 5 eases the proof of the correctness theorem, since the final value of that register is specified to be whatever the interpreter produces. This is of course not very helpful, but it allows us to follow the convention that we specify the entire resulting state while not bothering very much with irrelevant state elements.

We are now ready to state a theorem about the effect of executing the min subroutine on an **sm** state. The PVS terms `defs(s)` is the program memory and `halted(s)` is the halted flag of state `s`. The PVS terms `op(i)` and `arg1(i)` are the opcode and first argument of an instruction `i`. We use the constant `program` to represent the programs we wish to execute – it is an array of instructions that contains the min program listed in Figure 2.

```
min_correct: LEMMA
 op(defs(s)(pc(s))) = call
 AND arg1(defs(s)(pc(s)))= 0
 AND defs(s) = program AND NOT halted(s)
 AND 5 < R0(s) AND R0(s) <= R1(s) AND R1(s) < N
 =>
 sm(s, min_clock(s)) =
 goto(inc(pc(s)),
 write(2,R1(s),
 write(3,least(R1(s), R0(s), R0(s), mem(s)),
 write(4,0,
 write(5, min_correct_unspecified_R5(s),s)))))
```

2.3 Correctness Theorem Proof

PVS proofs of code correctness theorems like min_correct are relatively straightforward. We build the proof by proving lemmas about the constituent blocks. As suggested in the previous section, the structure of the clock functions guides the proof.

Straightline code is proved using a PVS strategy. The strategy expands to a PVS "grind" command that uses a standard set of lemmas applied as auto-rewrite rules to execute the code symbolically. Loops are proved using a second PVS strategy that expands into a sequence of PVS commands that set up the appropriate inductive argument and simplify as needed. Some lemmas about specification functions like `least` are typically needed for the proof to complete successfully. In particular, theorems pertaining to the type of the specification functions and how the specification functions relate to each other must be proved.

The use of PVS rewrite rules and PVS strategies aids the development of proofs about code execu-

```
address    code
  30       move 6 0
  31       move 5 1
  32       sub 5 6
  33       jumpz 5 42
  34       move 0 6
  35       call 0
  36       moverind 5 3
  37       moverind 4 6
  38       movewind 3 4
  39       movewind 6 5
  40       incr 6
  41       jump 31
  42       ret
```

Figure 3: **sm** sort subroutine

tion. By following our restrictive conventions about how to express code correctness lemmas these simple PVS strategies work quite well. However, even more importantly they provide a kind of proof resilience. Since the proofs are mostly automatic, modest changes to the code or specifications do not require that a wholly new proof be constructed.

2.4 A Second Example

We present a second example of an **sm** subroutine to emphasize that our approach is indeed largely automatic. Figure 3 presents a subroutine that sorts the array whose bounds are contained in registers 0 and 1. It is implemented using the min program described previously, and its proof is an example of how to build on other subroutine correctness theorems. We automate reasoning about code that calls subroutines as much as possible, and a subroutine correctness theorem of the form we have described does just that. By applying min_correct we can reason about **sm** code that calls min just as we reason about code that employs builtin **sm** instructions.

In order to specify the behavior of the sort subroutine, we define a function `sort` that sorts an array in the manner of the subroutine.

```
sort(cur, max, s): RECURSIVE state =
 if (cur < max)
 THEN let least=least(max, cur, cur, mem(s)) IN
       sort(cur+1,max,write(cur,mem(s)(least),
            write(least,mem(s)(cur),s)))
 ELSE s ENDIF
 MEASURE max(0,max-cur)
```

We prove a theorem about the effect of executing a sort subroutine call.

```
sort_correct: LEMMA
 op(defs(s)(pc(s))) = call
    AND arg1(defs(s)(pc(s)))= 30
    AND defs(s)=program AND NOT halted(s)
    AND 6<R0(s) AND R0(s)<=R1(s) AND R1(s)<N
 =>
 sm(s, sort_clock(s)) =
 goto(inc(pc(s)),
  write(0,if R0(s)<R1(s) THEN R1(s)-1
                         ELSE R0(s) ENDIF,
  write(2,if R0(s)<R1(s) THEN R1(s)
                         ELSE R2(s) ENDIF,
  write(3,sort_unspecified_R3(s),
  write(4,sort_unspecified_R4(s),
  write(5,0,
  write(6,if R0(s)<R1(s) THEN R1(s)
                         ELSE R0(s) ENDIF,
  sort(R0(s),R1(s),s))))))))
```

The proof of sort_correct has the same structure as the proof of min_correct. Each constituent block of code is specified and proved, and the PVS strategies for straightline and loop code are employed. The min_correct theorem is used to reason about the call to the min program, just as with any builtin **sm** opcode.

3 Reasoning About Specifications

Largely automatic proofs of programs as described in the previous section are robust in the sense we need them to be. A change to a program or even to the language semantics defined by the interpreter would require minimal changes in the proofs. However, these kinds of specifications are unsatisfactorily unclear and complex. The specifications reflect the algorithm used by the code and do not effectively convey the needed program functionality.

To use our mostly automatic approach in code proofs we limit ourselves by avoiding existential quantifiers and using a (primarily) first-order, recursive style. But our specification need not be so constrained. For example, what we really want to know about the sorting program is that it produces a sorted permutation of the original array without disturbing irrelevant memory elements. A good specification of the sorting program is:

```
sort_works: LEMMA
 op(defs(s)(pc(s))) = call
  AND arg1(defs(s)(pc(s)))= 30
  AND defs(s)=program AND NOT halted(s)
  AND 6<R0(s) AND R0(s)<=R1(s) AND R1(s)<N
  =>
 let s2= sm(s, sort_clock(s)) IN

% array is sorted
 (FORALL (i,j:subrange(R0(s),R1(s))):
   i<j => mem(s2)(i)<=mem(s2)(j)) AND

% array is permutation
 (EXISTS (f: (bijective?[subrange(R0(s),R1(s)),
                         subrange(R0(s),R1(s))])):
  (FORALL (i:subrange(R0(s),R1(s))):
    mem(s2)(f(i)) = mem(s)(i))) AND

% irrelevant memory is unchanged
 (FORALL (i: address): i>6 AND (i<R0(s) OR i>R1(s)) =>
   mem(s)(i)=mem(s2)(i))
```

The proof of this lemma is relatively straightforward, although like most PVS proofs involving the higher-order language capability of PVS the proofs are less automatic than the proofs in the last section. By applying the lemma proved in the previous section sort_correct, we reduce this theorem to one that does not involve our program or even – except hidden in the "unspecified" functions – the **sm** interpreter. We satisfy the remaining proof obligation involving desired properties and the specification function in conventional PVS proof style.

4 Conclusion

The automation of proofs about computer systems increases robustness and aids formal verification of realistic-sized computer systems. Recursively-defined interpreters can be used to define computer system behavior in a clean and simple way. The usefulness of these techniques transcends the particularities of different theorem proving systems.

Realistic proofs require robustness and PVS is capable of a proof style that fosters resilience in proofs about computer systems. The approach relies in part on restricting the manner in which we describe code execution, but the full PVS logic and prover may be used in concert with our automatic approach.

References

[1] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.

[2] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[4] Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

[5] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[6] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. 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.

[7] Warren A. Hunt, Jr. and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–47, Hemel Hempstead, UK, 1992. Prentice Hall International Series in Computer Science.

[8] Matthew Kaufmann and J S. Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–31, June 1996.

[9] 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, 1997. DRAFT.

[10] 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.

[11] J Strother Moore. *Piton – A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, 1996.

[12] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[13] 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.

[14] John Rushby. Personal Communication, November 1996.

[15] David M. Russinoff. A mechanically checked proof of IEEE compliance of the AMD K5 floating-point square root microcode. Available as www.onr.com/user/russ/david/fsqrt.html, August 1996.

[16] 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.

[17] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[18] Matthew Wilding. *Machine-Checked Real-Time System Verification*. PhD thesis, University of Texas at Austin, May 1996. Also available as ftp.cs.utexas.edu/pub/boyer/wilding-diss.ps.gz.

Acknowledgements I thank David Greve and David Hardin of the Advanced Technology Center for many suggestions that improved this paper considerably. This work was accomplished under the Rockwell Collins IR&D program.