# Towards Light-Weight Verification and Heavy-Weight Testing*

Stephan Pfab,[1], Harald Rueß,[2], Sam Owre,[2], Friedrich W. von Henke[1]

[1]Universität Ulm
Fakultät für Informatik
James-Franck-Ring
D-89069 Ulm
{vhenke,pfab}@informatik.uni-ulm.de

[2]SRI International
Computer Science Laboratory
333 Ravenswood Ave.
Menlo Park, CA, 94025
{owre,ruess}@csl.sri.com

**Abstract.** We give an overview on our approach to symbolic simulation in the PVS theorem prover and demonstrate its usage in the realm of validation by executing specification on incomplete data. In this way, one can test executable models for a possibly infinite class of test vectors with one run only. One of the main benefits of symbolic simulation in a theorem proving environment is enhanced productivity by early detection of errors and increased confidence in the specification itself.

## 1 Introduction

Traditional, simulation-based validation methods have not kept up with the scale or pace of modern digital design, and, therefore, form a major impediment in the drive for evermore complex designs (Hardin *et al.*, 1998). This is mainly due to the sheer number of possible test cases which makes it nearly impossible to perform exhaustive testing. Thus testing only demonstrates the existence but not the absence of flaws. Even worse, it is unlikely that testing alone would have caught errors like the famous bug in the lookup table of the Intel Pentium floating-point division unit, since it only occurred on table inputs that were thought to be beyond the region of interest (Pratt, 1995).

Formal verification methods based on theorem proving techniques, model-checking, or a combination thereof offer viable alternatives to simple testing, since formal methods permit proving the absence of errors (in the formal model). The construction of formal justifications, however, is usually at best semi-automatic for industrial-sized designs and the cost of doing formal analysis in an interactive way currently prevents formal methods from being integrated in the development cycle for both hardware and software.

* This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) project *Verifix* and by the Deutscher Akademischer Austauschdienst (DAAD) under the DAAD-NSF exchange program. The work undertaken at SRI was partially supported by the National Science Foundation under grant No. CCR-9509931. Published in: R. Berhammer, Y. Lakhnech (eds.), Tool Support for System Specification, Development and Verification, pp. 189–200 Advances in Computing Science, Springer, Wien, New York, 1999.

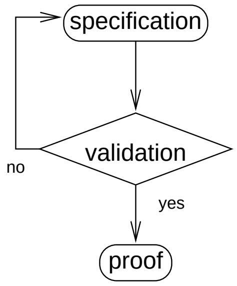


**Fig. 1.** Early Phase of Specification Life-Cycle

In many cases, however, formal models are *executable* and amenable to validation by means of simulation. Such an execution facility is required to support evaluation of partially specified models including uninterpreted constants or function symbols, i.e., *symbolic simulation*, in order to be widely applicable. It provides a useful capability that is intermediate between running individual test cases and exploring properties for all runs, since symbolic simulation on indeterminate data permits covering a class of cases—possibly of infinite cardinality—with one test run. Put in other words, *symbolic simulation* may be regarded as a hybrid of *light-weight* verification and *heavy-weight* testing. For this reason we take the liberty of using these terms interchangeably in this paper.

One of the main benefits of symbolic simulation in a theorem proving environment is enhanced productivity by early detection of errors and increased confidence in the specification, since symbolic evaluation permits investigating formal models fully automatically before resorting to formal proofs. This, somewhat idealized, view on the early phase of a specification life-cycle is depicted in Figure 1.

The applicability and usefulness of symbolic simulation in the ACL2 theorem prover (Kaufmann and Moore, 1997) has already been demonstrated (Moore, 1998). The ACL2 system inherits efficient simulation in a natural way from the underlying implementation language, since its logic of recursive functions is embedded in applicative Common Lisp. The situation is different for proof systems like Coq (Dowek *et al.*, 1993), PVS (Owre *et al.*, 1995), or Typelab (von Henke *et al.*, 1996), all of which are based on powerful static type systems that are beyond the capabilities of type systems of current programming languages. The PVS (Owre *et al.*, 1995) specification language, for example, relies on a rich set of typing constructs—like dependent types, predicate subtypes for expressing partial functions, and a powerful module system—in order to express formal models in a precise and succinct way. As a consequence, evaluation in systems with powerful static type systems cannot directly be inherited as in Acl2, and the reduction relation is usually implemented via substitution calculi. Although conceptually simple, these calculi are several orders of magnitude slower than specialized interpreters and compilers of off-the-shelf programming languages. This slowness has proved to be a formidable bottleneck for many medium- to large-size verification efforts such as the verification of pipelined microprocessors (Srivas and Miller, 1995, Stegmüller, 1998).

The purpose of this paper is to give an overview on our approach of integrating efficient symbolic simulation in the PVS theorem prover. The basic approach is to provide translations between PVS functions and Common Lisp in order to exploit the efficiency of Common Lisp for executing PVS functions. We demonstrate the usage of symbolic simulation in the realm of validating state-based models such as microprocessor specifications.

This paper is structured as follows. Section 2 describes our approach of symbolically executing PVS specifications. In order to make this paper largely self-contained, we recapitulate in Section 3 a basic method for encoding state machines in PVS; the running example is a model of a small, stack-based microprocessor. Section 4 demonstrates symbolic evaluation of state machines, and, finally, Section 5 closes with some remarks.

## 2 Efficient Symbolic Simulation

A certain subset of the PVS specification language (Owre *et al.*, 1998) can be regarded as an executable, functional programming language that includes various operations on expressions of basic types, conditionals (`if`, `cases`, `cond`, `table`), total recursive functions defined by means of measure recursion, and structural recursion on abstract datatypes (catamorphisms, paramorphisms). Here we describe an extension to PVS for (symbolically) evaluating programs with Lisp-like speed.

We use the idea of *inverse evaluation* (Berger and Schwichtenberg, 1991) and compute a normal form for a functional PVS expression in three successive steps: first, an expression is translated into the corresponding Lisp expression (and compiled to machine code using the Lisp compiler); second, the Lisp program is executed using Lisp's evaluation function `eval`; finally, the result of evaluation is translated back to a corresponding PVS expression.

This process of retranslation includes the translation of Lisp closures to PVS $\lambda$-expressions by generating a bound PVS variable, by evaluating the closure on the Lisp translation of this variable, and by retranslating the result to the PVS level. In addition to the technique described in (Berger and Schwichtenberg, 1991) we support evaluation of abstract datatype expressions and include the compilation of recursively defined PVS functions—which are required by the type system of PVS to be total. Altogether, we obtain Lisp-like execution speed for normalizing functional PVS expressions including uninterpreted constants and function symbols, since the Lisp compiler can readily be used to generate machine code.

Obviously, evaluation of programs that include uninterpreted constant and function symbols yields boolean conditions that evaluate to neither `true` nor `false` (e.g. `IF (x + f(2) < 2 * x) THEN` $e_1$ `ELSE` $e_2$ `ENDIF`). In these cases we make use of the PVS prover to simplify or, whenever possible, to decide such formulas (including quantification). For the expressiveness of the full PVS logic, however, there cannot be a single proof procedure for deciding all kinds of formulas. Therefore, our symbolic evaluator `norm` is parameterized with respect to a prover strategy in order to simplify expressions in a problem-specific way. This functionality is realized in the Lisp compilation `if*` of PVS conditionals.

1

```
 fac(m: nat): RECURSIVE nat =
   IF m <= 1 THEN 1 ELSE m * fac(m - 1) ENDIF
MEASURE m
```

Consider the simple example of symbolically evaluating `fac(n + 2)`, where the factorial function `fac` in 1 is specified as a recursive PVS function and `n` is an unknown natural number.

```
(norm "fac(n + 2)" :strategy (assert))
   -->  (n + 2) * IF n + 1 <= 1 THEN 1 ELSE (n + 1) * fac(n) ENDIF
```

The expression to be evaluated is presented as a string to the simplifier `norm` and the strategy argument is used in prover calls. Consequently, all PVS strategies, including user-defined ones, are acceptable arguments to `norm`. In our example, the strategy `(assert)` causes this evaluator to use the PVS decision procedures to simplify conditionals, and the prover is—not too surprisingly—able to decide that `n + 2 <= 1` does not hold (since `n` is of type `nat`). But, without further information about `n` from the current context, one cannot decide whether `n+1 <= 1` is true; thus, evaluation stops at this point.

## 3 State Machines

This section describes an often-used method for describing state machines in PVS (see also (Srivas *et al.*, 1997)). As a simple example we use a transcription of the stack machine from (Boyer and Moore, 1996). Furthermore, we sketch the generation of Lisp code for symbolically evaluating this machine.

The specification of the stack machine is packaged in a theory that is parameterized with respect to the length `N` of the memory array. All naturals less than `N` are valid addresses, the enumeration type `opcodes` in 2 lists the opcodes of the machine, and the record type `instr` determines the format of instructions.

2

```
address: TYPE = below[N]

opcodes: TYPE = {MOVE, MOVEI, MOVEWIND, MOVERIND, ADD, SUB,
                 INCR, DECR, JUMP, JUMPZ, CALL, RET, HALT}

instr  : TYPE = [# op: opcodes, arg1, arg2: address #]
```

States consist of a program counter, a stack, the memory, a flag for halting the processor, and the program code. Both the memory array and the program array are modeled as finite functions, and states are represented as elements of the record type `state`.

```
                                                           3
state: TYPE = [#    pc: address,
                   stk: list[address],
                   mem: [address -> nat],
                halted: bool,
                  code: [address -> instr] #]
```

The five state components are accessed by `pc`, `stk`, `mem`, `halted`, and `code`, respectively.

Individual instructions are given semantics by defining *state transformers* that appropriately modify the current state `s` of the machine. Functions and records may be "modified" in PVS by means of an update expression. The result of an update expression is a function or record that is exactly the same as the original, except that at the specified arguments it takes the new values. For example, the move instruction with indirect addressing `movewind` applied to addresses `a1`, `a2` increments the program counter (modulo `N`) and updates the memory at address `mem(s)(a1)` with the value `mem(s)(a2)` if the location to be updated is valid; otherwise the machine is halted.

```
                                                           4
movewind(a1, a2: address)(s: state): state =
  IF mem(s)(a1) < N THEN
    s WITH[ pc := inc(pc(s)),
           mem := mem(s) WITH[mem(s)(a1) := mem(s)(a2)]]
  ELSE
    s WITH [halted := TRUE]
  ENDIF
```

Recall that `mem(s)` is a function with domain `address`. Likewise, `movewind(a1, a2)` is a (curried) function from states to states. Given such a function for every instruction, a one-step interpreter `execute` for the stack-machine is defined by case analysis on the opcode of the given instruction.

```
                                                                    5
exec1(i: instr): [state -> state] =
  LET a1 = arg1(i), a2 = arg2(i) IN
     CASES op(i) OF
       MOVE    : move(a1, a2),
       MOVEI   : movei(a1, a2),
       MOVEWIND: movewind(a1, a2),
       ...,
       JUMP    : goto(a1),
       HALT    : halt
     ENDCASES
```

Using the techniques described in Section 2, the memory component `mem(s)`, for example, is compiled to the following Lisp function (the prefix "pvs::" indicates PVS functions and types, and the memory size `N` is instantiated to 30 in this case).

```
(defun mem (s)
  (if (vectorp s)
    (svref s 2)
    #'(lambda (x)
        (wrap (pvs::make-application
                 (pvs::make-field-application '|mem| (unwrap s))
                 (phi x pvs::address[30]))))))
```

If the argument of this function is a Lisp vector then one simply uses the `Lisp` vector lookup `svref`. Otherwise, the argument is uninterpreted and we compute a closure by wrapping a PVS application. The function `phi` retranslates Lisp terms (here: a Lisp symbol) to PVS expressions (here: a variable of type `pvs::address[30]`). Similarly, the `movewind` function in $\boxed{4}$ translates to Lisp code that is structurally similar to the corresponding PVS function, whereby symbol names ending in '$*$' indicate symbolic versions of the corresponding Lisp functions and predicates.

```
(defun movewind (a b)
  #'(lambda (s)
      (if* (<* (funcall (mem s) a) 30)
        (vector (code s)
                (halted s)
                #'(lambda (x)
                    (if* (=* x (funcall (mem s) a))
                       (funcall (mem s) b)
                       (funcall (mem s) x)))
                 (inc (pc s))
                 (stk s))
        (vector (code s) t (mem s) (pc s) (stk s)))))
```

Records are currently translated to Lisp vectors, the PVS $\lambda$-expression `mem(s)` is realized as a Lisp closure, memory lookup `mem(s)(a)` translates to the Lisp function application `(funcall (mem s) a)`, and memory lookup at the Lisp level is simply encoded as function overwrite.

This naive translation scheme has the disadvantage that new state vectors are allocated in every simulation step of the machine. Even worse, the number of conditionals to be decided for array lookup depends on the number of updates of the array, with the disastrous consequence that considerably fewer than 100 instructions of the stack machine can be simulated on a workstation (Sun Ultra 1).

Modern compilation technology such as structural analysis or monads (Jones and Wadler, 1993) would be helpful to guarantee *single-threadedness* and, consequently, to use in-place updates in a safe way. Implementation of structural analysis, however, is non-trivial and monads require a specialized specification style. Currently, we only support simple runtime tests to ensure safe in-place updates. More specifically, finite functions of type `[below[n] -> A]`, where `n` is a positive integer and `A` an arbitrary type, are translated to a structure containing a tag and a vector for representing the finite function (array) under consideration. A second tag stored in the vector is used by the functions `lookup` and a destructive version of `update` to ensure safe in-place updates. The function update creates a new structure with a new tag and a shallow copy of the vector which is modified to hold the new data and the new tag. If the program is not single-threaded then one of `lookup` or `update` detects a mismatch of the tags and aborts at run-time.[2]

Consequently, with this approach it is safe to use in-place updates at the expense of some runtime overhead. The specification of the stack machine above, for example, is single-threaded, and the use of destructive updates yields run times that are several orders of magnitude faster (roughly a factor of 1000) than with the naive approach described above.

## 4 Symbolically Simulating State Machines

In this section some characteristic features of symbolic simulation in PVS are demonstrated using the running example of computing the minimum element in an array with the stack machine in the previous section. The semantics of this machine has been given in terms of a one-step interpreter, and the machine's basic cyclic behavior `exec(s, n)` is then defined by iterating the one-step interpreter `exec1` on the state `s`. Here, the integer argument `n` serves as an upper bound on the number of steps, in order to guarantee termination (all functions in PVS are required to be provably total). Such an interpreter is formalized in the module `exec` in $\boxed{6}$ in a machine-independent way. This module is parameterized

[2] Instead of simply aborting execution, one could also use a recovery technique. We have implemented such a strategy by storing update-index-tag triples in a hash table; preliminary experiments suggest that this recovery strategy roughly doubles run times.

by a state type, a one-step interpreter, and a predicate for characterizing abort states. In addition, the parameter `observe` can be used to observe the dynamic behavior of the interesting parts of states.

```
                                                                        6
exec[
    state: TYPE,
     step: [state -> state],
  halted?: pred[state],
        A: TYPE,
  observe: [state -> A]
]: THEORY
BEGIN

 exec_rec(n: nat, s: state, acc: list[A]): RECURSIVE list[A] =
   IF n = 0 OR halted?(s) THEN reverse(acc)
     ELSE LET s1 = step(s), newacc = cons(observe(s1), acc) IN
            exec_rec(n - 1, s1, newacc) ENDIF
 MEASURE n

 exec(max: nat, s: state): list[A] =
   exec_rec(max, s, null)

END exec
```

The recursive function `exec_rec` in $\boxed{6}$ iterates the step function and accumulates the observable part of resulting states; the type system of PVS together with the `MEASURE` ensures that this function is total. One advantage of our approach of evaluating PVS functions at the Lisp level lies in the fact that a Common Lisp compiler can readily be used to eliminate, for efficiency reasons, tail-recursive calls like in `exec_rec`.

A particular interpreter for the stack machine is obtained by instantiating the module with actual parameters. The fourth and the fifth parameter below indicate that the complete state is being observed.

```
                                                                        7
 sm: THEORY = exec[state, step, halted?,
                   state, (LAMBDA (s: state): s)]
```

The program `MIN` in $\boxed{8}$ for computing the index of the minimum element in an array is used as an example. Since the format of all instructions is fixed to contain exactly two arguments, an uninterpreted family `X(i)` of "don't cares" models unused argument positions. The `loader` function simply constructs an array from the more convenient list notation of programs; its definition is not given here.

```
                                                                    8
X: [nat -> address]

MIN: ARRAY[address -> instr] =
  loader(0, (: (MOVE    , 2   , 0),         % mem[2]:= mem[0]
               (MOVE    , 3   , 0),         % mem[3]:= mem[0]
               (MOVE    , 4   , 1),         % mem[4]:= mem[1]
               (SUB     , 4   , 2),         % mem[4]:= mem[4] - mem[2]
               (JUMPZ   , 4   , 12),        % if mem[4]=0 then pc := 12
               (INCR    , 2   , X(0)),      % mem[2]:= mem[2]+1
               (MOVERIND, 4   , 2),         % mem[4]:= mem[mem[2]]
               (MOVERIND, 5   , 3),         % mem[5]:= mem[mem[3]]
               (SUB     , 5   , 4),         % mem[4]:= mem[4] - mem[2]
               (JUMPZ   , 5   , 2),         % if mem[5]=0 then pc := 2
               (MOVE    , 3   , 2),         % mem[3]:= mem[2]
               (JUMP    , 2   , X(1)),      % pc    := 2
               (RET     , X(2), X(3)) :))   % pc    := ‘top of stack’
```

If called with two addresses `i` and `j` in memory locations `0` and `1`, the program `MIN` leaves the address of the minimum content of the array from `i` through `j` in memory location `2`. Consider the following state `S1`.

```
STK: list[address]; MEM: [address -> nat]

S1: state =
 (# pc     := 0,
    stk    := STK,
    mem    := MEM WITH
               [(0)  := 6,    (1) :=  15,  (6) := 102,  (7) := 111,
                (8)  := 103,  (9) := 103, (10) := 103, (11) := 101,
                (12) := 103, (13) := 103, (14) := 101, (15) := 103],
    halted := false,
    code   := MIN #)
```

The stack component is uninterpreted and all memory locations except for a finite number are “don’t cares”. If we (symbolically) evaluate the stack machine interpreter `sm.exec` in $\boxed{6}$ on the argument `(100, S1)` in PVS we get a list of `73` states. The first entry of this list represents the final state of the computation. Thus `car(sm.exec(100, S1))` evaluates to:

```
                                                                9
(# code   := MIN,
   halted := true,
   mem    := LAMBDA (X_33: address[N]):
               IF X_33 = 2 THEN 15
               ELSE IF X_33 = 4 THEN 1
               ...
               ELSE IF X_33 = 3 THEN 11
               ...
               ENDIF,
   pc     := 6,
   stk    := STK1 #)
```

In order to observe the dynamic behavior of the memory location `mem(s)(3)` one produces a new interpreter `smo.exec` by instantiating the generic interpreter `exec` in the following way.

```
                                                                10
smo: THEORY = exec[state, step, halted?,
                   int, (LAMBDA (s: state): mem(s)(3))]
```

Now, symbolic evaluation of `smo.exec(100, S1)` yields a list of the values of the result location during evaluation.

```
(: 11, ..., 11, 6, ..., 6, X(3) :)
```

Symbolic evaluation builds up huge expressions and soon becomes unmanageable when a large number of conditions evaluate neither to `true` nor `false`.

```
                                                                11
MEM2: [nat -> address]

MEM2_ax: AXIOM
  FORALL(n: (nat | n /= 6)): MEM2(n) > MEM2(6)

S2 : state = (# pc     := 0,
                stk    := null,
                mem    := MEM2 WITH [(0) := 5, (1) := 20],
                halted := false,
                code   := MIN #)
```

Simple evaluation of `car(smo.exec(100, S2))`, for example, essentially yields unfolded expression trees, since conditions like `MEM2(12) < MEM2(6)` or `MEM2(14) = MEM2(6)` can not be decided by evaluation but only by use of the axiom `MEM2_ax` $\boxed{11}$. In these situations the symbolic evaluator calls the PVS prover with the current context as hypotheses, the condition to be decided as the proof goal, and a suitably defined proof strategy; hereby, arbitrary strategies—including user-defined tactics—of the PVS prover can be used to decide conditions. For the example above, it suffices to apply a tactic that finds appropriate instances of the axiom `MEM2_ax` followed by a call to the PVS decision procedures in order to simplify the expression `car(smo.exec(100, S2))` to get the desired outcome `6`.

## 5 Conclusion

We have described a symbolic simulator for a functional subset of the PVS specification language and demonstrated its usage for validating a simple assembler program for a stack machine by applying it to incomplete data.

The main characteristics of our symbolic evaluator are its efficiency due to compilation of PVS functions to LISP, retranslations of LISP closures to PVS lambda-expressions, and the usage of theorem proving to decide conditions involving uninterpreted constants, uninterpreted function symbols, or even quantified expressions.

The effectiveness of symbolically evaluating PVS specifications has been demonstrated through validation of a number of small to medium-sized case studies. Besides the toy stack machine (Wilding, 1997) described in this paper we have used symbolic evaluation of specifications to validate PVS models of microprocessors like the Transputer or the (pipelined) DLX (Börger and Mazzanti, 1996, Stegmüller, 1998). Further application of symbolic simulation include animation of the denotational semantics of imperative programs (Pfeifer *et al.*, 1996), validation of bisimulation diagrams, and partial evaluation of functions and state machines. Moreover, our evaluator has not only been used for the validation of formal models but has also shown to be useful in the context of theorem proving itself. Efficient evaluation proved to be a necessity for safely extending theorem proving capabilities by replacing deduction with the evaluation of meta programs (von Henke *et al.*, 1998). Furthermore, a variant of our simulator has recently been added to the main simplification strategy of the PVS prover (Shankar, 1998).

Transcriptions of abstract state machine (ASM) (Gurevich, 1995) models like the ones described in (Börger and Mazzanti, 1996) into PVS demonstrate that symbolic evaluation can readily be used to animate deterministic ASMs. Basically, so-called *dynamic functions* are translated to state transformers, and a centralized case analysis, like the one for stack machine above, is used to dispatch the ASM rules. Since the majority of ASM specifications we have encountered can easily be rewritten in this way, our symbolic simulator may form a viable alternative to specialized ASM simulators.

Although symbolic evaluation of PVS specifications has proven to be quite effective for our case studies—it executes around 100000 stack machine instructions per second on a Sun Ultra 1—it may not be efficient enough for animating myriads of test cases for large state machines like industrially-sized microprocessors. To further improve efficiency of symbolic evaluation, it is necessary develop a compiler from PVS to Lisp that uses the power of the PVS type system in a systematic way to produce more efficient code; for example, by avoiding `bignums`. Equally important, an interactive environment for PVS—including the usual infrastructure like tracing, stack inspection, breakpointing, and statistical and diagnostic information—is essential for effectively exploring expression values during execution.

**Acknowledgements.** The initial transcription of the stack machine to PVS has been performed by J. Rushby and M. Wilding sent us his modified specifications. We also thank H. Schwichtenberg for providing us with his implementation of inverse evaluation.

## References

U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed $\lambda$-calculus. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

E. Börger and S. Mazzanti. A Correctness Proof for Pipelining in RISC Architectures. Technical Report DIMACS 96-22, Dipartimento di Informatica, University of Pisa, Corso Italia 40, 56125 Pisa, Italy, 1996.

R.S. Boyer and J S. Moore. Mechanized Formal Reasoning about Programs and Computing Machines. In *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Murthy, C. Parent, Chr. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide (Version 5.8)*. INRIA-Rocquencourt – CNRS - ENS Lyon, 1993. Projet Formel.

Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 9–36. Computer Science Press, 1995.

D. Hardin, M. Wilding, and D. Greve. Transforming the Theorem Prover into a Digital Design Tool: From Concept Car to Off-Road Vehicle. In *CAV'98: Computer-Aided Verification*, Lecture Notes in Computer Science. Springer Verlag, June 1998.

S.L. Peyton Jones and Ph. Wadler. Imperative Functional Programming. In *ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, Charleston, January 1993.

M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 21(4):203–213, 1997.

J S. Moore. Symbolic Simulation: an ACL2 Approach. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, Lecture Notes in Computer Science. Springer Verlag, 1998. Accepted for publication.

S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

S. Owre, N. Shankar, J.M. Rushby, and D.W.J Stringer-Calvert. *The PVS Specification Language, Version 2.2*. Computer Science Lab, SRI International, Menlo Park CA 94025, September 1998. From: `http://www.csl.sri.com/pvs.html`.

H. Pfeifer, A. Dold, F. W. v. Henke, and H. Rueß. Mechanized Semantics of Simple Imperative Programming Constructs. Ulmer Informatik-Berichte 96-11, Universität Ulm, Fakultät für Informatik, 1996.

V. Pratt. Anatomy of the Pentium Bug. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 97–107. Springer Verlag, May 1995.

N. Shankar. Personal communication. 1998.

M.K. Srivas and S.P. Miller. Formal Verification of the AAMP5 Microprocessor. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

M. Srivas, H. Rueß, and D. Cyrluk. Hardware Verification using PVS. In Th. Kropf, editor, *Formal Hardware Verification Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, chapter 4, pages 156–205. Springer Verlag, 1997.

M. Stegmüller. Formale Verifikation des DLX RISC-Prozessors: Eine Fallstudie basierend auf abstrakten Zustandsmaschinen. Master's thesis, Universität Ulm, 1998. From `http://www.informatik.uni-ulm.de/ki/Edu/Diplomarbeiten`.

F.W. von Henke, M. Luther, H. Pfeifer, H. Rueß, D. Schwier, M. Strecker, and M. Wagner. The TYPELAB specification and verification environment. In M. Nivat M. Wirsing, editor, *Proceedings AMAST'96*, pages 604–607. Springer LNCS 1101, 1996.

Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case Studies in Meta-Level Theorem Proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, September 1998.

M. Wilding. Robust Computer System Proofs in PVS. Presented at LFM'97: the Fourth NASA Langley Formal Methods Workshop; also available from `http://www.csl.sri.com/sri-csl-fm.html`, 1997.