# A Grand Challenge Proposal for Formal Methods: A Verified Stack

J Strother Moore

Department of Computer Sciences, University of Texas at Austin,
Taylor Hall 2.124, Austin, Texas 78712
`moore@cs.utexas.edu` telephone: 512 471 9590
WWW home page: `http://www.cs.utexas.edu/users/moore`

**Abstract.** We propose a grand challenge for the formal methods community: build and mechanically verify a practical embedded system, from transistors to software. We propose that each group within the formal methods community design and verify, by the methods appropriate to that group, an embedded system of their choice. The point is not to have just one integrated formal method or just one verified application, but to encourage groups to develop the techniques and methodologies necessary for system-level verification.

**Keywords**: hardware verification, software verification, simulation, modeling, theorem proving, model checking

## 1 The Challenge

Each major group in the formal methods community should design and mechanically verify a practical embedded system, from transistors to software.

## 2 Why the CLI Stack Is Not Enough

In the late 1980s, Computational Logic, Inc. (CLI), with support from DARPA and others, built and verified the "CLI stack" [3] using the Boyer-Moore theorem prover, Nqthm [5]. The major layers in the stack were:

- a register transfer level (RTL) design for a microprocessor
- an operational semantics for a machine code instruction set architecture (ISA),
- an operational semantics for a relocatable, stack-based assembly language
- operational semantics for two simple high-level languages, one based on Gypsy (a derivative of Pascal) and one based on Lisp,
- some application programs, including a program to play winning Nim, and
- a simple operating system.

Connecting these layers were various functions. The "downward" functions were

- a function that maps from ISA states to RTL states,
- an assembler, linker and loader that, when composed, map from the assembly language states to ISA states, and
- compilers that map from the high-level language states to the assembly language states.

In addition, partial “upward” maps were also defined, allowing us to speak of the high-level data represented by a region of a low-level state. The obvious theorems relating these layers and maps were all proved mechanically. Finally, pre- and post-conditions for the applications programs were defined and the programs were proved correct with respect to the high-level language semantics.

These theorems were then composed into a main theorem that may be paraphrased as follows:

> **Theorem.**
> Suppose one has a high-level language state satisfying the pre-condition of a given application program. Construct the RTL state obtained from the high-level state by successively compiling, assembling, linking, and loading. Run the RTL machine on the constructed state. Map certain data regions up, using the partial inverse functions, producing some high-level data objects. Then the post-condition holds of that high-level data.

In the 1989 description of the stack, the microprocessor at the base was the FM8502 (a 32-bit version of the 16-bit FM8501 [13]), whose ISA was designed, implemented at the gate-level, and verified by Warren Hunt. The assembly language was named Piton [22] and the assembler, linker, and loader were designed, implemented and verified by J Moore, with help from Matt Kaufmann. The two high-level languages, micro-Gypsy[34] and a subset of the Nqthm Pure Lisp[9] and their compilers were implemented and verified by Bill Young and Art Flatau, respectively. The operating system, KIT [2], was designed, implemented, and verified by Bill Bevier. The NIM application [33] was implemented and verified by Matt Wilding.

In 1992, Bishop Brock and Warren Hunt formalized the Netlist Description Language (NDL) of LSI Logic, Inc. They used this formal NDL to describe a new microprocessor, the FM9001, and verified that the new design implemented the new ISA. Unlike the FM8501 and FM8502, the FM9001 was fabricated (by LSI Logic, Inc.) [14]. Moore then ported the Piton assembly language to the FM9001 ISA, by changing and reverifying the code generators. The rest of the stack was then ported painlessly and automatically. The porting process is described in some detail in [22]. Thus, by 1992, a functioning verified stack existed.

This description makes it seem as though the challenge was met a decade ago. How did the CLI stack fall short? There are many reasons.

- The FM9001 was modeled at the gate level.
- The FM9001 had memory-mapped io and a crude interrupt facility because one of the sponsors did not want a useful trusted microprocessor competing with the product of a much larger project.
- The FM9001 had an unrealistically simple architecture: no pipeline, no speculation, no floating point, no cache. (It might not be necessary to include all these features in the next verified stack; it depends on the particulars of the stack chosen.)
- The FM9001 was fabricated on a gate-array.
- None of the programming languages described had io facilities, for obvious reasons.
- The high-level languages were too simple to be of practical use. For example, the micro-Gypsy language had very few primitives and no dynamically allocated data types (e.g., records). The Pure Lisp had automatic storage allocation (e.g., `cons`) but no verified garbage collector.
- The assemblers and compilers were “cross-platform” transformers: they did not run on the FM9001 but on commercial machines. For example, using a Sun workstation running Nqthm (written in Common Lisp) it was possible to produce, from a high-level micro-Gypsy state, a binary image for the FM9001. But it was not possible to produce this image using only an FM9001, much less was it possible to produce it *in situ*. In addition to implementing the full Nqthm programming language and a verified garbage collector, one would have to implement and verify the parsers and a file system.

– No useful applications programs were verified.
– The operating system was not ported to the FM9001 (because it was not actually implemented for the FM8501 but for a different, concurrently developed machine).

It should also be noted that the FM8502 and Piton work were in part responsible for another major Nqthm proof effort carried out in the late 1980s and early 1990s: Yuan Yu's modeling of the Motorola 68020 and his verification of 21 of the 22 Berkeley C String Library subroutines [6]. Yu verified these subroutines by verifying the binary machine code produced by `gcc -o`, using the basic techniques developed by the Nqthm community to prove assembly code programs correct [4].

## 3 Why It May Be Practical Today

The CLI stack was a *technology driver* for the Boyer-Moore community. Two important products came from the stack work.

The first was the formalized operational semantics methodology. This methodology is characterized by a certain style of definition for state machines, a certain collection of lemmas to control the symbolic simplification of successive state transitions, the use of clock functions to specify how long the paths of interest are, the formalization of commuting diagrams and how to "stack" them, and the understanding of the role of successive refinement as formalized by subtly different operational models. While most of these ideas pre-date the Boyer-Moore community, their formal expression and careful orchestration came to fruition in that community, and the CLI stack was the ultimate demonstration.

The second product of the CLI stack research was an industrial strength version of the Nqthm theorem prover, ACL2 [16]. ACL2 was started in 1989 and was ready for its first industrial trials by 1992. So the design of ACL2 was directly influenced by the CLI stack.

The stack forced us to confront such issues as:

– the enormous size of both the formal models and the proofs relating them: ACL2 contains many modified heuristics from Nqthm, permitting ACL2 to deal with much larger functions and data bases;
– execution efficiency of formal models: ACL2 supports applicative Common Lisp as the logic to leverage the work of others on Lisp compilation and execution efficiency;
– syntactic succinctness: ACL2 supports macros where Nqthm only supports functions;
– building on the work of others and collaborative proofs: ACL2 supports books of definitions and theorems proved by others and allows the incremental extension of the rule data base, where Nqthm supported only the saving and restoring of the entire data base;
– proof structuring and name and rule scoping: ACL2 supports encapsulation, local environments, and packages;
– types as a specification mechanism: ACL2 supports guards and can mechanically insure type correctness with respect to the declared guards; furthermore, ACL2 rewards type correctness with faster execution; and
– goal-specific hints: ACL2 permits the user to give goal-specific hints, easy swapping in and out of various rule theories, and computed hints.

These are but a few of the major departures ACL2 made from Nqthm in response to the difficulty of dealing with the CLI stack.

With ACL2, parts of commercial applications have been verified.

- The Motorola CAP digital signal processor was modeled at the pipelined architecture level and the sequential microcode ISA level, with bit- and cycle-accurate models. The second model executed several times faster than Motorola's own simulator. These ACL2 models were proved equivalent under the hypothesis that no pipeline hazards were present in the code sequence being executed. The hazard predicate was defined as an executable ACL2 function on the microcode ROM. Hazards not identified by the architects were identified during the equivalence proof. In addition, the ROM was found to contain programs with hazards, despite the fact that the Motorola microcode programmers had been forewarned of some of the hazard conditions. Two microcode programs were proved correct. See [?,8].
- The pseudo-code representing the microcode of the kernel of the `FDIV` floating-point divide operation on the Advanced Micro Devices K5 was verified to implement the IEEE 754 floating point standard. See [24].
- The register-transfer level (RTL) circuit descriptions for all the elementary floating-point arithmetic on the AMD Athlon have been proved to implement the IEEE 754 floating point standard. The ACL2 functions modeling the RTL circuits – functions produced by a mechanical translator – were executed on over 80 million floating point test vectors from AMD's standard test suite and found to produce exactly the same output as AMD's RTL simulator; thus ACL2's executability was crucial to corroborating the translation scheme against AMD's accepted, if informal, RTL "semantics," its C simulator. Upon attempting to prove that the ACL2 functions implemented floating point arithmetic correctly, bugs were found. This is important since the RTL had been extensively tested. The bugs were fixed by modifying the RTL. The modified models were verified mechanically. All of this happened at AMD and before the Athlon was first fabricated. See [27, 28].
- The Java Virtual Machine was modeled and fabricated by Rockwell Collins Avionics and manufactured by aJile Systems, Inc. The ACL2 model executed at 90% of the speed of Rockwell's previously written C simulator for the design and became the standard test bench for the chip design. See [12, 11].
- An academic microprocessor, the FM9801, with speculative execution, exceptions and interrupts was verified to implement an ISA. See [30, 29].
- A security model for the IBM 4758 secure co-processor was produced and analyzed to obtain FIPS-140 Level 4 certification. See [31].
- A safety-critical compiler-checker was verified for train-borne real-time control software written by Union Switch and Signal. See [1].
- A checker for the Ivy theorem prover from Argonne National Labs was verified to be sound. Ivy proofs are thus generated by unverified code but confirmed to be proofs by a verified ACL2 function. See [21].
- Proof methods have been developed whereby ACL2 can be used to verify that one state machine is a stuttering bisimulation of another. After so verifying the appropriateness of a finite-state abstraction of an infinite-state system, for example, it is possible to use a verified ACL2 model checker [19] to verify the correctness of the finite-state machine and then infer the correctness of the infinite-state system. See [18, 20]
- A BDD package was verified to be sound and complete with ACL2. This package uses ACL2's single-threaded objects to achieve runtime speeds of about 60% those of the CUDD package. However, unlike CUDD, the verified package does not support dynamic variable reordering and is thus more limited in scope. See [32].
- A operational JVM thread model was produced and Java code was verified by verifying the bytecode produced by the `javac` compiler of Sun Microsystems, Inc. The particular Java application creates an unbounded number of threads in contention for a single object in the heap. The correctness proof required establishing mutual exclusion. See [25, 26].

These projects all made extensive use of the features added to ACL2 in response to the CLI stack: heuristics for handling huge models, execution efficiency, macros, books, encapsulation, packages, guards, and goal-specific hints. These projects raise to a new level the complexity of the hardware and software designs verified. Many of these projects were done either for industry or by industrial researchers; all of these projects have had impact on both the utility of formal methods and on the perception of that utility. Had CLI not embarked on the CLI stack project in the mid-1980s, had it confined its attention to what Nqthm was traditionally being used for (number theory, meta-mathematics, algorithms), it is doubtful that ACL2 would have been created.

In addition, the CLI stack has had impact on other verification groups. In particular, the Pro-Cos project [17] was directly inspired by the CLI stack and focused the work of many European verification groups. One in particular, the VERIFIX group [10] centered in Kiel and Karlsruhe, Germany, has focused extensively on the compiler verification problem. Thanks to that work we now understand much better how to host the verified stack on the verified microprocessor.

The projects cited above also contribute to our understanding of how to build a realistic verified stack. If you re-read the list with a stack in mind, you will see that various components missing from the CLI stack are now present in some limited form.

We can now produce a more elaborate and realistic microprocessor, complete with floating point and interrupts. We can hope to verify important microcode. We can develop proof-objects and the associated verified proof checkers that let us verify boot-level components on unverified machines and then check the proofs on certified machines. Our JVM model looks strikingly like the Piton assembly language and there is reason to expect we could produce a verified JVM compiler and that we could routinely verify Java code by verifying JVM bytecode.

In short, the work done with ACL2 in the last ten years has positioned the ACL2 community well for tackling a more ambitious verified stack along the following lines:

– a small, low-power modern microprocessor,
– a compiler from JVM bytecode to native machine code,
– a JVM runtime system including a garbage collector and native methods for io, and
– a Java application, e.g., an cryptosystem, smartcard transaction interface, etc.

The advantage of hosting the JVM on the device is that it invites further verified extensions. One obvious extension is a verified proof checker, which could in turn be used to check proof produced by an untrusted theorem prover.

But the stack described above is just a suggestion for the ACL2 community's attack on this challenge. Each group should choose a suitable stack.

## 4 Why It is Hard

However, building a verified embedded system – from transistorrs to software – is still an enormous undertaking. Evidence of that is as follows.

– Increasingly, we are seeing power and clock controlled by software. A modern verified microprocessor should address this issue, in both the hardware design and its specification. We are unaware of any formal work in this area.
– No one has verified the netlist description of a modern microprocessor. The closest anybody has come is Sawada's FM9801 work, which was formalized one level higher than a netlist and is missing many features like the realistic use of caches and bus protocols.
– We are unaware of any practical work on the verification of input/output routines or the verification of code employing them.

- We are unaware of any practical work on the verification of a garbage collector implementation in the context of a practical programming language execution environment.
- Our JVM work does not address exceptions or dynamic class loading.
- We are unaware of any practical work on the verification of a practical concurrent system interacting with a shared memory as rich as the JVM heap. Our own work on the use of "chaotic" uniprocessors to verify concurrent programs [23] is a start.
- We are unaware of any formal modeling of io devices, e.g., LCD screens or keypads, network and disk interfaces, etc., and their integration with other components of the system. This will raise questions of asynchronous communication.
- No one has verified a large body of code in a formally specified language since Yu verified the Berkeley C String Library.

The basic reason is that it is hard to verify complex systems. It is obvious that no one will use formal methods to do system-level verification if the task remains this hard.

The only way I can see to make it less hard is to do it, repeatedly, until doing it is mechanical. This actually happens. For example, Yuan Yu eventually reached the point that he could verify simple MC68020 machine code programs mechanically about as fast as he could formalize their specification and loop invariants. Similarly, researchers at both AMD and at Rockwell Collins can now verify RTL designs and certain ISA relationships with comparable facility. The reason is not just understanding the problem but the construction of special-purpose tools to highlight the creative parts and mechanize the mechanical.

In addition to the sheer complexity of the challenge, it is hard for another reason. It is awkward in many formal systems to span multiple layers of abstraction, as is required in a verified stack.

One problem with this proposal is that it may not be clear that the individual parts of the verified stack are worth a Ph.D. Perhaps all that is left is engineering. This is a problem since it is likely that the workforce needed to carry out such a proposal will consist primarily of graduate students. I have three things to say about this. First, the nature of a grand challenge is that one must be able to see a path to success. Otherwise, we would just adopt open problems and hope someone comes up with the key insight. Second, I personally have faith that there are many Ph.D. dissertations yet to be found in this work. Many ordinary people build correct systems; but verifying them seems to require extraordinary talent. Why? I conjecture that it is because we have not really formalized the abstractions the original authors are exploiting. I know of no other way to uncover these abstractions than to try to verify practical systems and reflect on why we "know" they are correct. Third, I observe that some choices of the embedded system clearly lead to a Ph.D., e.g., produce a verified theorem prover as the application running atop the verified stack.

One might ask: why not just formalize a single layer of a system and explore the abstraction and verification issues there? That is what we have been doing for the past ten years. My response is simple: it is too easy to sweep the hard part of a problem under the rug if you are free to define the starting point arbitrarily. An implemented stack forces a concrete starting point and all else must be handled by deduction.

## 5 The Proposed Rules of the Game

This proposal does not suggest we all verify the same embedded system, nor does it suggest we all work together. It suggests that each group choose an embedded system and verify it from gates to software. I do not believe the time is right to try to build a single, integrated formal system, as would be necessary if we were to all work together collaboratively. Nor do I believe it is scientifically advantageous for us all to be verifying the same system: no such challenge could include all the important unsolved problems.

But I do suggest a few "rules."

- The project should produce an artifact, e.g., a chip.
- The artifact's behavior should be of interest to people not in formal methods.
- The artifact should come with a "warranty" expressed as a mathematical theorem.
- The warranty should be certified mechanically; user input and interaction are allowed but a machine must be responsible for the soundness of the claim.
- The machine used to certify the warranty should be available for others, at least, to use and test, if not inspect.

The rules disallow the CLI stack, because the artifact's behavior was not otherwise interesting.

The rules are meant to disallow the following: Group A uses their tool to prove or check a theorem in their logic. Group B uses their tool to prove or check a theorem in their logic. Then, the two theorems are cited and used to claim that the warranty is a theorem. The problem with this "solution" is that no mechanical device – theorem prover, model checker, SAT solver, whatever – actually verifies that the two theorems can be composed appropriately, because they are theorems in two different logics. In order to claim success, someone would have to build a mechanical interface between the two logics and argue that it was done soundly. I do not mean to discourage collaboration between groups or the use of a wide variety of tools. Quite the contrary. But the rules governing when a formula in one logic justifies the validity of a formula in another must be formalized and mechanized and the mechanization must be available for others to use. Verification of a vertically integrated stack tends to encourage the development of mechanically supported logical frameworks so that appropriate logical and mathematical support is available at each layer and the results of successive layers can be combined to get a system-level main result.

## 6 What This Challenge Would Encourage in ACL2

How would ACL2 evolve if the ACL2 community undertook this challenge? Put another way, what is needed to make ACL2 more effective for system-level verification?

I can think of many improvements and so will list only a few.

ACL2 does not always use the available computing cycles well. In particular, only big symbolic evaluation proofs are really aided by faster machines. Most little theorems on which ACL2 failed ten years ago still fail, even though orders of magnitude more cycles are expended on them. But to do system verification we must allow the machine to carry more of the burden. That is, the user should have the reasonable expectation that waiting longer or throwing more machines at the problem will help ACL2 find the proof. This means that ACL2 must be programmed to do more search, but in a non-combinatoric way. One suggestion is the use of several different top-level strategies, tried in series or in parallel (see below). Another is the use of examples to guide search. A third is the use of proof plans, which in ACL2 might best be explored by the less constrained use of forcing.

ACL2 would benefit greatly by building and exploiting a parallel execution engine. The ACL2 programming language is applicative; it affords the opportunity for massive parallelism. A parallel execution engine for ACL2 could produce a real breakthrough. At Rockwell Collins and AMD we are already seeing bit- and cycle-accurate models of commercial components. These models run at near C speeds and permit formal analysis. These models get execution efficiency by using ACL2's single-threaded objects, which support a sequential programming model. But what if a parallel ACL2 execution engine would permit applicative formal models to execute orders of magnitude faster than conventional C simulators? A parallel execution engine would confer another benefit: since the theorem prover is coded in ACL2, it too would run faster. Many parts of the theorem prover could be embarrassingly parallel. When a goal splits into $n$ subgoals, each must be proved and the proofs are independent. If we had $n$ processors, each running ACL2 with the same books loaded, the cost of farming out these $n$ subgoals would be just the cost of writing and reading the $n$ formulas. Communication in the other direction would consist simply of a single bit indicating success or failure. We have many proofs in which $n$ is 1000 or more. Non-combinatoric search (above) would make it possible to use many more processors productively.

The CAP DSP proofs produced intermediate formulas requiring 25 megabytes to print. Such subgoals could be "understood" only with the aid of tools such as Emacs and `grep`. System verification routinely produces such big formulas and we need an interface that allows the user to understand what he or she is seeing. We are currently developing visualization tools for large proofs. We need to develop interactive steering that does not come at the expense of automatic search.

The fundamental scientific hypothesis of the ACL2 project is that it is possible to build books (files of definitions and lemmas) that make it practical to reason about realistic applications. We need to construct better books. Without doubt, the single most important ACL2 book for system verification is a machine arithmetic book. By machine arithmetic we mean integer arithmetic in which virtually every operation produces a floor or mod expression. We have been working on this book for many years. This ties directly into the idea of making ACL2 use cycles to search: arithmetic frequently requires the adoption of several different normal forms and proof styles. For example, to prove an equality one sometimes simplifies both sides to the same expression. But one sometimes proves two weak inequalities instead. Similarly, one sometimes distributes multiplication over addition, producing "polynomial normal form." But in other proofs one factors. We imagine different books coding these different strategies and trying them all on the top level goal. This is not combinatoric. We have already developed several such search strategies, including the swapping in of a new set of rules after the formula has stabilized under simplification, the use of context sensitive metafunctions to code the search for polynomial normal forms and cancellation, and the addition of heuristics for non-linear arithmetic.

We also need better books for bit vectors – especially the view of machine arithmetic as bit vector manipulation.

We need to exploit algorithms from model checking and finite state machine analysis in general. For example, if an ACL2 theorem is finite state, i.e., can be proved by doing computation on a finite number of cases, then it might be worthwhile simply to consider the cases. It is likely that many of the more tedious proofs of control logic in the FM9001 were finite state and could have been knocked out this way without human intervention.

At the level of code proofs for programs in languages defined operationally, we need several tools. They include the automatic generation of weakest preconditions and the metatheorem that allows the proof of a theorem about the operational semantics via theorems about the weakest preconditions. Put another way, we need a verified weakest precondition generator for any language we define operationally. We need tools for generating "clock functions" measuring the lengths of certain computation paths. We need tools for generating the functional equivalent of a program's semantics and automatically proving the theorem that the functions are correct. We need tools for doing symbolic simulation through a realistically large interpretive semantics.

We will need tools to help infer generalizations, abstractions and inductive invariants in the commonly occurring situations. These activities are especially burdensome when one is doing code proofs. We are working on the abstraction problem now and have plans to look at the automatic strengthening of proposed invariants.

To verify a stack we will have different people working concurrently on different parts of the proof. The inter-office communication in collaborative proofs is extremely disruptive, both because users find it necessary to amend the definitions and lemmas that define the interfaces between components and because users find it helpful to share lemmas. We need ways to manage concurrent proof efforts. One obvious tool is a search engine that crawls over the known books on the web and imports theorems of possible relevance. Another is the use of data mining techniques to identify closely connected cliques of function symbols and thus automatically limit search to the theorems in certain cliques – marking rules that leap from one clique to another so that they are tried experimentally rather than routinely. If rules are to be imported from other users, we will need tools to help orient rewrite rules and to help complete the data base of rules, *a la* Knuth-Bendix completion, but heuristic in nature since our rules are conditional and can be recursive.

Finally, to host the verified compilers on a verified microprocessor we will need ways to check proofs that are produced by untrusted tools. This leads to the idea of producing formal proofs or "proof objects" and checking them with trusted proof checkers. Indeed, proof objects are even liable to play a role in the interactive steering and visualization tools, since they completely record the trace of the proof. They are fundamental in the construction of a trusted theorem prover.

This survey of likely areas of future research in the ACL2 community justifies my faith that there are many Ph.D. dissertations in the next verified stack.

## 7 Why it is Important

The verified stack project will be a technology driver for formal methods at a time when such methods are comfortably settling into niches much smaller than they have the potential to fill.

In the Boyer-Moore community, the CLI stack solidified our understanding and formal use of operational semantics, simplification of state machines, refinement, etc. That methodology was developed because the stack forced vertical integration of complicated layers – layers so complicated that the failure to properly abstract and specify the interfaces would doom the attempt to stack correctly. I believe that the adoption of this challenge will severely strain the vertical integration support in any of today's verification systems.

The specification challenge here is extreme. One of the major problems in the CLI stack was specifying each level adequately so as to provide the guarantees needed by the next highest level. Among the problems are resource bounds, errors, and non-termination. Designing the stack interfaces will undoubtedly improve our ability to specify realistic components.

I also believe that the sheer magnitude of the job of building a practical embedded system will force the development of tools to mechanize the job. Several times in my research career I have almost reached the point of giving up and building tools to mechanize what I was doing – only to find that the difficulty of building the needed tools was greater than that of the remaining tasks. So I just "toughed it out" – and never built the tools that would have made it easier in the future. I do not believe this challenge will yield to the "tough it out" strategy.

I believe that by building on a solid basis in which the underlying assumptions are clear and by building with components that are well specified, the time to build new components will decrease. That is, I believe that when formal methods are widely used throughout a project and when the basic "folklore" of a project is formalized into re-usable books, time-to-market will be decreased.

I believe that building an embedded system with correctness as a major criterion for success will force the hardware and software designers to design for (relative) simplicity, elegance, and modularity at a time when all other forces on them push them toward unmanageable complexity. I believe this push for clarity will be beneficial beyond the embedded system designed.

Finally, if successful, this challenge will put on the table a practical verified trusted system, the very existence of which will both enable and inspire the development of more ambitious ones.

## References

1. P. Bertoli and P. Traverso. Design verification of a safety-critical embedded verifier. In Kaufmann et al. [15], pages 233–246.
2. W. R. Bevier. A verified operating system kernel. Ph.d. dissertation, University of Texas at Austin, 1987.
3. W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.
4. R. S. Boyer and J S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176. MIT Press, 1996.
5. R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, New York, 1997.
6. 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.
7. B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, LNCS 1166, November 1996. `http://www.cs.utexas.edu/users/moore/-publications/bkm96.ps.Z`.
8. B. Brock and J S. Moore. A mechanically checked proof of a comparator sort algorithm, 1999. `http://-www.cs.utexas.edu/users/moore/publications/csort/main.ps.Z`.
9. A. D. Flatau. A verified implementation of an applicative language with dynamic storage allocation. Phd thesis, University of Texas at Austin, 1992.
10. W. Goerigk and U. Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In *Proceedings FM-TRENDS'98 International Workshop on Current Trends in Applied Formal Methods*, LNCS, Boppard, 1998.
11. D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In Kaufmann et al. [15], pages 113–136.
12. David A. Greve. Symbolic simulation of the JEM1 microprocessor. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design – FMCAD*, LNCS 1522. Springer-Verlag, 1998.
13. W. A. Hunt. *FM8501: A Verified Microprocessor.* Springer-Verlag LNAI 795, 1994.
14. W.A. Hunt and B. Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, April 1992.
15. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, 2000.
16. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, 2000.
17. H. Langmaack. The ProCoS Approach to Correct Systems. *Real Time Systems*, 13:253–275, 1997.
18. P. Manolios. Correctness of pipelined machines. In *Formal Methods in Computer-Aided Design, FMCAD 2000*, pages 161–178. Springer-Verlag LNCS 1954, 2000.
19. P. Manolios. Mu-calculus model-checking. In Kaufmann et al. [15], pages 93–112.
20. P. Manolios, K. Namjoshi, and R. Sumners. Linking theorem proving and model-checking with well-founded bisimulation. In *Computed Aided Verification, CAV '99*, pages 369–379. Springer-Verlag LNCS 1633, 1999.
21. W. McCune and O. Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In Kaufmann et al. [15], pages 265–282.
22. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language.* Automated Reasoning Series, Kluwer Academic Publishers, 1996.

23. J S. Moore. A mechanically checked proof of a multiprocessor result via a uniprocessor view. *Formal Methods in System Design*, 14(2):213–228, March 1999.
24. J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.
25. J S. Moore and G. Porter. An executable formal JVM thread model. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*. USENIX, April 2001. `http://www.cs.utexas.edu/users/-moore/publications/m4/model.ps.gz`.
26. J S. Moore and G. Porter. The apprentice challenge. *TOPLAS*, (accepted for publication, 2002). `http://www.cs.utexas.edu/users/moore/publications/m5/index.html`.
27. D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. http://www.onr.com/-user/russ/david/k7-div-sqrt.html.
28. D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In Kaufmann et al. [15], pages 201–232.
29. J. Sawada. Verification of a simple pipelined machine model. In Kaufmann et al. [15], pages 137–150.
30. J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *Computed Aided Verification, CAV '98*, pages 135–146. Springer-Verlag LNCS 1427, 1998.
31. S. W. Smith and V. Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *The Third USENIX Workshop on Electronic Commerce*, September 1998.
32. R. Sumners. Correctness proof of a BDD manager in the context of satisfiability checking. In *Proceedings of ACL2 Workshop 2000*. Department of Computer Sciences, Technical Report TR-00-29, November 2000. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/final/sumners2/paper.ps`.
33. 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. See URL ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/wilding-cav93.ps.
34. W. D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Comp. Logic. Inc., Austin, Texas, 1988.