A Mechanized Program Verifier

J. Strother Moore

Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, USA
moore@cs.utexas.edu,
http://cs.utexas.edu/users/moore

Abstract. In my view, the "verification problem" is the theorem proving problem, restricted to a computational logic. My approach is: adopt a functional programming language, build a general purpose formal reasoning engine around it, integrate it into a program and proof development environment, and apply it to model and verify a wide variety of computing artifacts, usually modeled operationally within the functional programming language. Everything done in this approach is software verification since the models are runnable programs in a subset of an ANSI standard programming language (Common Lisp). But this approach is of interest to proponents of other approaches (e.g., verification of procedural programs or synthesis) because of the nature of the mathematics of computing. I summarize the progress so far using this approach, sketch the key research challenges ahead and describe my vision of the role and shape of a useful verification system.

1 Approach

The verification community seeks to build a system that can follow the reasoning of an average human engaged in a creative, computational, problem solving task lasting months or years and spanning an arbitrary set of application domains.

Now step back and reconsider what was just said: we seek to build a system that follows – if not reconstructs or anticipates – the reasoning of an average human in an arbitrary, creative, computational, problem solving task. This is the theorem proving problem, perhaps restricted to a computational logic. We seek to build a system that reasons about computation.

I believe one is not likely to achieve a goal unless one identifies precisely what the goal is. My goal is build a practical theorem prover for a computational logic. I believe that any attack on the verification problem will fail unless theorem proving – machine assisted reasoning – is at its heart.

My[1] approach to the "verification problem" is thus:

[1] This vision is consistent with McCarthy's, was developed by Boyer and me and then Kaufmann and me, and probably describes the vision of verification as seen by most of the "Boyer-Moore" community. But the community has not been consulted. So I use the first person pronoun here.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 268–276, 2008.

- Adopt a functional programming language so that programs are functions in a mathematical theory. The particular language I use is the functional subset of the ANSI standard programming language Common Lisp.
- The theory is described by a set of axioms and rules of inference, including well-founded induction and a definitional principle that allows conservative introduction of new concepts.
- The theory is directly supported by an interactive theorem prover.
- The theorem prover employs heuristics and various decision procedures so that many "simple" proofs can be found completely automatically.
- The theorem prover is designed to operate automatically once a conjecture is posed to it. Its behavior is determined by previously proved theorems residing in its database. This addresses three key problems:

  - Since verifying functional correctness of interesting programs requires operation in an undecidable domain, provision for some user guidance is unavoidable. It should not be an after-thought, nor should it force the user to eschew powerful automatic features.
  - Using previously proved lemmas to guide the system encourages the user to think at the high level, i.e., about concepts involved in the specification and their relationships. This also encourages the creation of libraries of general concepts and lemmas.
  - Automatic operation (with respect to previously proved theorems) facilitates *proof maintenance*: the task of verifying a system produced by making incremental modifications to a previously verified one.

- The programming language and theorem prover are embedded in a program/proof development environment in which prototyping, testing, and proving are seamlessly integrated. Like any good programming environment, ours supports a rich collection of code/proof structuring tools including name scopes, modules, libraries, etc., in which the work of many other developers can be made available. The theory must support such tools formally.
- While everything proved in this system exemplifies software verification – the mathematical language is an ANSI standard programming language – systems written in other languages may be modeled and verified by using the system as a formal meta-language. This allows many different languages to be related within a common framework.
- To demonstrate the practicality of a functional programming language the system should be implemented in it.
- To keep the work focused on the goal, every opportunity should be taken to verify systems of commercial (or at least, outside) interest.

A prototype of this vision of mechanized verification was first demonstrated by Boyer and Moore in 1973 in what was called "The Edinburgh Pure Lisp Theorem Prover." The demonstration has been continuously improved and elaborated through a series of so-called "Boyer-Moore theorem provers," including Nqthm [8] and ACL2 (by Kaufmann and Moore, with early contributions by Boyer) [21].

2 Progress So Far

Here is a chronology of work done by the Boyer-Moore community. This represents 35 years of unbroken pursuit of the software verification grand challenge. This litany makes plausible the vision I describe later.

- **1970–1979.** Fully automatic proof of insertion sort, including the permutation property, and fully automatic proofs of many other theorems about Pure Lisp programs [3]; the correctness of a McCarthy-Painter-like expression compiler, the soundness and completeness of a propositional tautology checker, and the correctness of the Boyer-Moore fast string searching algorithm in FORTRAN 77 [4]; proof of the correctness of a linear-time majority vote program written in FORTRAN 77 [7]; proof that a "cruise control" program keeps a vehicle on course in a smoothly varying cross-wind and homes to the course if the wind becomes steady [10]. During this period, Boyer and Moore worked on the Software Implemented Fault Tolerance (SIFT) project [15] and our attempts to formalize the BDX 930 to explain the mix of Pascal and machine code in that system drove much of the subsequent improvement to the theorem prover.
- **1980–1989.** The invertibility of the RSA encryption algorithm [6]; the unsolvability of the halting problem [5]; Gödel's First Incompleteness Theorem [34]; the correctness of a gate-level microprocessor design [19]; the correctness of an operating system kernel [1]; the correctness of an assembler, linker and loader for a stack based relocatable symbolic assembly language supporting Booleans, integer arithmetic, bit vectors, arrays, and recursive subroutine calls – the system produced binary images for the microprocessor mentioned earlier and the proof guaranteed functional correctness of the binary images with respect to the machine code ISA [2]; the correctness of a compiler from a subset of Pascal to the assembly code above and the verification of some simple applications written in that language [38]; the correctness of a compiler from a subset of Pure Lisp to the assembly code [14]; the composition of many of the above theorems to make it possible to prove an application program correct with the high-level language semantics and then derive in one step the correctness of its binary image under the gate-level semantics of the microprocessor – – this is known as the *verified stack* of Computational Logic, Inc., and it was completed and published in a special issue of the *Journal of Automated Reasoning* in 1989.
- **1990–1999.** Proof that an NDL netlist implements the machine code ISA of a 32-bit microprocessor and the fabrication of the microprocessor by LSI Logic [20]; porting the verified stack to the fabricated machine by re-targeting and re-verifying the assembler [27]; verification that a Nim-playing program plays winning Nim and the demonstration of this program on a fabricated, verified microprocessor using the verified stack [37]; verification of 21 of the 22 routines in the Berkeley Unix C String Library – performed by compiling the library with `gcc -o` to obtain binary machine code for the Motorola 68020 and then verifying that with respect to a formal operational semantics capturing 80% of the user-level 68020 instructions [11]; proof by the same

technique of a variety of other C programs, including the C code for binary search and Hoare's *in situ* Quick Sort from [24]; proof that the microarchitectural design of the Motorola CAP digital signal processor (dsp) implements a certain microcode engine, including the verification that a pipeline hazard detection algorithm was sufficient to insure bit- and cycle-accurate equivalence of the two models [12]; use of the formal dsp microcode engine as a simulator for the microarchitecture, because the formal microcode model was three times faster than Motorola's SPW model of the microarchitecture [12]; proof of several dsp microcode programs written by Motorola [12,13]; proof that the microcode for the AMD K5 correctly implemented IEEE floating point division – carried out *before* the K5 was fabricated [29]; proof that all elementary floating point on the AMD Athlon was correctly implemented in RTL (a variant of Verilog)[32]; proof of the soundness of a Lisp program that checks the proofs produced by the Ivy theorem prover from Argonne National Labs – Ivy proofs may thus be generated by unverified code but confirmed to be proofs by a verified Lisp function [26]; proof that a security model of the IBM 4758 secure co-processor satisfied properties required for FIPS 140-1 Level Four certification [35]; development and production use of a formal model at Rockwell Collins as the microarchitectural simulator for the first silicon-implemented JVM (the design became the JEM1 of aJile Systems, Inc.) [18] – the formal simulator runs at 90% of the speed of a comparable simulator written in C.

– **2000–2005.** Proofs of properties of components of the AMD Opteron and other processors [private communication]; proof of the soundness and completeness of a Lisp implementation of a BDD package that achieves 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) [36]; proof of correctness of the algorithms used for floating point division and square root on the IBM Power 4 [33]; proof of instruction equivalence between different implementations of a commercial microprocessor [16]; proof that microcode for the Rockwell Collins AAMP7 implements a given security policy having to do with process separation [17]; verification that the JVM bytecode produced by the Sun compiler `javac` on certain simple Java classes implements the claimed functionality [28], including verification of a small class file that spawns an unbounded number of non-terminating threads in contention for a common data structure [30]; verification of certain properties of the Sun bytecode verifier as described in JSR-139 for J2ME JVMs [25] (part of an ongoing effort to verify the runtime safety of the JVM).

Many other applications are available at [23].

3 Research Challenges and Milestones

Kaufmann and I describe the our research challenges in [22]. These include the mechanized invention of lemmas and new concepts, including the discovery of

inductive invariants (perhaps by augmenting a user supplied core invariant); the use of examples and counterexamples to guide search and concept and conjecture formation; analogy, learning and data mining in theorem proving; the adoption of an open architecture for a theorem prover allowing it to build on other work and to be tailored by the user in a sound way; support for parallel and collaborative theorem proving projects; an empowering interactive user interface supporting, among other things, interactive steering of an ongoing proof attempt; training people to use these tools; the construction of a useful and verified theorem prover hosted on a verified platform.

4 Discussion and Speculation

While I strongly advocate and actively work on the integration of decision procedures and static analyzers into mechanized theorem provers to ease the burden of proof, I do not believe they are the breakthroughs needed to make software verification palatable to the masses. I do not believe software verification will ever be palatable to the masses (until the AI challenge is solved).

I believe that mechanized verification of the functional correctness of software crucially depends upon the designer or some other human annotating the code to explain the intention of important routines or blocks. This will not happen until programmers and their masters stop measuring productivity in lines-of-code per day and start insisting on functional correctness as a deliverable. This will probably never be commonplace because most software is non-critical.

Nevertheless, for critical applications the software industry ought to have a way to check the correctness of its products.

What follows is my own speculation as to the verification system of the future. The references below point to closely related work mentioned above. Scrutiny of those references will support my conviction that this proposal is plausible.

The necessary tool suite will have to be tightly integrated with several programming languages to provide the necessary assurance, runtime efficiency and proof power. Code portability will be provided by a virtual machine (VM). Two levels of programming language are provided. The high-level language will be a functional one – and that same language will be the mathematical language in which proofs are conducted. [The entire Boyer-Moore project supports the conclusion that adequate speed can be obtained via a functional language while providing a unified framework for machine-aided reasoning.] A verified compiler will map that language into the low level language, which will be the assembly code of the VM [27,14]. The formal semantics of the VM are given operationally in the logic [25]. Thus, VM code can be verified, using the techniques of [31] to mix inductive assertion-style proofs with direct proofs. Special static analyzers, especially something akin to escape analysis or the restrictions enforced by ACL2 on single-threaded objects [9] (ACL2's version of monads), will allow the mixture of functional high-level programs with occasional calls to VM code for efficiency.

The entire programming environment is integrated into a theorem proving environment [21]. This keeps the user focused on the obligation to deliver correct

code and eliminates cognitive dissonance. The theorem prover will make our current systems seem weak and rigid. It will: be fully automatic but steered by context; provide visualization and animation of the proof search process so the user understands what is happening; be capable of using a vast database of examples to guide search, concept formation, and conjecturing; and be parallelized so that multiple strategies can be pursued simultaneously.

5 Summary

I believe the "verification problem" is the theorem proving problem, restricted to a computational logic. Are we likely to build a system that follows and reconstructs human reasoning if we adopt a lesser goal?

References

1. Bevier, W.R.: A verified operating system kernel. Ph.d. dissertation, University of Texas at Austin (1987)
2. Bevier, W.R., Hunt, W.A., Moore, J.S., Young, W.D.: Special issue on system verification. Journal of Automated Reasoning 5(4), 409–530 (1989)
3. Boyer, R.S., Moore, J.S.: Proving theorems about pure lisp functions. J. ACM 22(1), 129–144 (1975)
4. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, New York (1979)
5. Boyer, R.S., Moore, J.S.: A mechanical proof of the unsolvability of the halting problem. Journal of the Association for Computing Machinery 31(3), 441–458 (1984)
6. Boyer, R.S., Moore, J.S.: Proof checking the rsa public key encryption algorithm. American Mathematical Monthly 91(3), 181–189 (1984)
7. Boyer, R.S., Moore, J.S.: Mjrty – a fast majority vote algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe, pp. 105–117. Kluwer Academic Publishers, Automated Reasoning Series, Dordrecht (1991)
8. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook, Second Edition. Academic Press, New York (1997)
9. Boyer, R.S., Moore, J.S.: Single-threaded objects in ACL2. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, Springer, Heidelberg (2002), `http://www.cs.utexas.edu/users/moore/publications/stobj/main.ps.gz`
10. Boyer, R.S., Moore, J.S., Green, M.W.: The use of a formal simulator to verify a simple real time control program. In: Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra, pp. 54–66. Springer, Heidelberg (1990) (Texts and Monographs in Computer Science)
11. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. Journal of the ACM 43(1), 166–192 (1996)
12. Brock, B., Hunt Jr., W.A.: Formal analysis of the motorola CAP DSP. In: Industrial-Strength Formal Methods, Springer, Heidelberg (1999)
13. Brock, B., Moore, J.S.: A mechanically checked proof of a comparator sort algorithm. In: Engineering Theories of Software Intensive Systems, IOS Press, Amsterdam (to appear, 2005)

14. Flatau, A.D.: A verified implementation of an applicative language with dynamic storage allocation. Ph.d. thesis, University of Texas at Austin (1992)
15. Goldberg, J., Kautz, W., Mellear-Smith, P.M., Green, M., Levitt, K., Schwartz, R., Weinstock, C.: Development and analysis of the software implemented fault-tolerance (sift) computer. Technical Report NASA Contractor Report 172146, NASA Langley Research Center, Hampton, VA (1984)
16. Greve, D., Wilding, M.: Evaluatable, high-assurance microprocessors. In: NSA High-Confidence Systems and Software Conference (HCSS), Linthicum, MD (March 2002), http://hokiepokie.org/docs/hcss02/proceedings.pdf
17. Greve, D., Wilding, M.: A separation kernel formal security policy (2003)
18. Greve, D.A.: Symbolic simulation of the JEM1 microprocessor. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, pp. 203–203. Springer, Heidelberg (1998)
19. Hunt, W.A.: FM8501: A Verified Microprocessor. LNCS, vol. 795. Springer, Heidelberg (1994)
20. Hunt, W.A., Brock, B.: A formal HDL and its use in the FM9001 verification. In: Proceedings of the Royal Society (April 1992)
21. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston (2000)
22. Kaufmann, M., Moore, J.S.: Some key research problems in automated theorem proving for hardware and software verification. Revista de la Real Academia de Ciencias (RACSAM) 98(1), 181–196 (2004)
23. Kaufmann, M., Moore, J.S.: The ACL2 home page. In: Dept. of Computer Sciences, University of Texas at Austin (2006), http://www.cs.utexas.edu/users/moore/acl2/
24. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, Second Edition. Prentice Hall, Englewood Cliff (1988)
25. Liu, H., Moore, J.S.: Executable JVM model for analytical reasoning: A study. In: Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME 2003), San Diego, CA, ACM SIGPLAN, New York (2003)
26. McCune, W., Shumsky, O.: Ivy: A preprocessor and proof checker for first-order logic. In: Kaufmann, M., Manolios, P., Moore, J.S. (eds.) Computer-Aided Reasoning: ACL2 Case Studies, Boston, MA, pp. 265–282. Kluwer Academic Press, Dordrecht (2000)
27. Moore, J.S.: Piton: A Mechanically Verified Assembly-Level Language. In: Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht (1996)
28. Moore, J.S.: Proving theorems about Java and the JVM with ACL2. In: Broy, M., Pizka, M. (eds.) Models, Algebras and Logic of Engineering Software, pp. 227–290. IOS Press, Amsterdam (2003), http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03
29. Moore, J.S., Lynch, T., Kaufmann, M.: 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 (1998)
30. Moore, J.S., Porter, G.: The Apprentice challenge. ACM TOPLAS 24(3), 1–24 (2002)
31. Ray, S., Moore, J.S.: Proof styles in operational semantics. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 67–81. Springer, Heidelberg (2004)

32. Russinoff, D.: 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 (1998), http://www.onr.com/user/russ/david/k7-div-sqrt.html
33. Sawada, J.: Formal verification of divide and square root algorithms using series calculation. In: Proceedings of the ACL2 Workshop, 2002 (April 2002), http://www.cs.utexas.edu/users/moore/acl2/workshop-2002
34. Shankar, N.: Metamathematics, Machines, and Godel's Proof. Cambridge University Press, Cambridge (1994)
35. Smith, S.W., Austel, V.: Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In: The Third USENIX Workshop on Electronic Commerce (September 1998)
36. Sumners, R.: 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
37. Wilding, M.: Nim game proof. Technical Report CLI Tech Report 249, Computational Logic, Inc. (November 1991), http://www.cs.utexas.edu/users/boyer/ftp/nqthm/nqthm-1992/examples/numbers/nim.eventsq
38. Young, W.D.: A verified code generator for a subset of Gypsy. Technical Report 33, Comp. Logic. Inc. Austin, Texas (1988)

A Discussion on J. Strother Moore's Presentation

Kathi Fisler

In both the introduction to your talk and your list of open problems, there are suggestions of strong connections to artificial intelligence. The AI researchers that I know, all believe that the foundations of AI are now statistical more than logical. Are we being short-sighted, if we look at this as a logical problem?

J Moore

I do not think so, because of the constraint I have posited, which was: We are trying to reason about programs. I agree that trying to reason about the world in general is beyond the scope of that book; but I believe trying to reason about programs is within the scope. I could not agree more with John Rushby about the fact that in the end, system verification requires arguing about the physical world, and I am explicitly not trying to do that. I am trying to stay focused on something that I believe is achievable and is amenable to a formal approach.

Greg Nelson

Jay, in discussing the *Clink* stack, you commented that you believe that software verification inherently has to handle multiple levels of abstraction. A comment

with which I deeply agree. In my work, I have aimed at different kinds of abstraction layers. There are a lot of abstractions, of course, between a *while*-loop and the RTL [register transfer level], and it is enormously impressive you are able to do that stuff. But there are equally many layers of abstraction between the *print()* of "Hello world" and the bitmap that finally actually puts the bits into the window. And most of my work has been on building verification tools that can properly and with modular soundness reason about the abstraction layers in object-oriented systems in higher levels of the software. They definitely spin abstraction layers. And I guess my question is, whether you think that I am missing anything in working at multiple abstraction layers, but the higher layers?

J Moore

No, I agree. I feel, it is almost about abstraction, and we should recognize that. But unfortunately, a lot of work on abstraction recognizes it by dealing what I would guard as toy abstractions, and not actually the practical abstractions we use, like compilers and the semantics of languages like Java, which are powerful abstractions in themselves. And the whole idea of *print()*. One of the interesting things that I did at computational logic was actually implementing a printer that puts out a bitmap, and spend some time thinking about the question: How do I know, that bitmap has got an "A" in it? So, yes, I believe that that level of abstraction is just as important as the ones that we explicitly dealt with in the stack. And all of them have to be dealt with in order to follow the kind of reasoning, a programmer engages in constantly. And that is the real challenge in my view.