Industrial Proofs with ACL2

Matt Kaufmann[1] and J Strother Moore[2]

[1] Advanced Micro Devices, Inc., 5900 East Ben White Blvd., Austin, TX 78741,
matt.kaufmann@amd.com
[2] Department of Computer Sciences, University of Texas at Austin,
Taylor Hall 2.124, Austin, Texas 78712,
moore@cs.utexas.edu

1 Abstract

Today it is impractical to prove – formally and mechanically – the correctness of entire computing systems of commercial interest. There are many reasons for this, both technical and economic. Mechanized theorem proving is nevertheless relevant in commercial hardware and software production. But practical considerations require that we focus our attention on problems that are both technically feasible within the time limits available and of interest to system designers. In this note we describe a few such industrial problems addressed with ACL2. We conclude with a brief description of what ACL2 is and how it is used.

Keywords: ACL2, hardware verification, software verification, formal verification, theorem proving, automated reasoning.

2 Applications

"ACL2" is the name of a functional programming language (based on Common Lisp), a first-order mathematical logic, and a mechanical theorem prover. The theorem prover is used to prove theorems in the logic – theorems about functions defined in the programming language.

"ACL2" stands for A Computational Logic for Applicative Common Lisp.

ACL2, which is sometimes called an "industrial strength version of the Boyer-Moore system," is the product of Kaufmann and Moore, with many early design contributions by Boyer. The ACL2 theorem prover is interactive in the sense that the user is responsible for the strategy used in proofs. But it is automatic in the sense that once started on a problem, it proceeds without human assistance. In the hands of an experienced user, the theorem prover can produce proofs of complicated theorems.

It has been used for a variety of important formal methods projects of industrial and commercial interest, including:

– verification that the register-transfer level description of the AMD Athlon^{TM} processor's elementary floating point arithmetic circuitry implements the

IEEE floating point standard [11, 12]; similar work has been done for components of the AMD K5 processor [10], the IBM Power 4 [13], and the AMD Opteron^{TM} processor.[1]

– verification that a microarchitectural model of a Motorola digital signal processor (DSP) implements a given microcode engine [1] and verification that specific microcode extracted from the ROM implements certain DSP algorithms [2];
– verification that microcode for the Rockwell Collins AAMP7 implements a given security policy having to do with process separation [3];
– verification that the JVM bytecode produced by the Sun compiler `javac` on certain simple Java classes implements the claimed functionality [9] and the verification of properties of importance to the Sun bytecode verifier as described in JSR-139 for J2ME JVMs [7];
– verification of the soundness and completeness of a Lisp implementation of a BDD package that has achieved 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) [15];
– verification 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 [8].

Other applications are described in [4] and in the papers distributed as part of the periodic ACL2 workshops, the proceedings of which may be found via the Workshops link on the ACL2 home page [6]. All papers in the ACL2 Workshop series are available online and each is accompanied by supporting material that allows you to reproduce the results in the paper with ACL2. Thus, the Workshops constitute a large body of examples illustrating how different users attack different kinds of problems. We recommend browsing the Workshop papers for work that seems similar to the work you might be doing with ACL2.

As these examples demonstrate, it is possible to construct mechanically checked proofs of properties of great interest in industrial hardware and software designers. The properties proved are typically not complete characterizations of the correctness of the systems studied. For example, the proofs about the AMD microprocessors – the AMD K5 processor, the AMD Athlon processor, and the AMD Opteron processor – just deal with the IEEE compliance of certain floating point operations modeled at the register transfer level. The microprocessors contain many unverified components and the verified ones could fail due to violations of their input conditions.

Nevertheless, these theorems were proved for good reason: the designers were concerned about their designs. Aspects of these designs are quite subtle or complicated and formal specification and mechanized proof offer the most complete way to relieve the concerns that something critical to correct functionality had been overlooked in the designs.

[1] AMD, the AMD logo, AMD Athlon, AMD Opteron, and combinations thereof, are trademarks of Advanced Micro Devices, Inc.

In addition to being interesting, these theorems are hard to prove. That is a relative judgment of course. Compared to longstanding open problems, these theorems are all trivial. But by many measures each of these proofs is much more complicated than any proof ever encountered by most readers. For example, the IEEE compliance proof for the floating point division microcode for the AMD K5 processor (in 1995) required the formal statement and proof of approximately 1,200 lemmas. Subsequent AMD floating-point proofs are harder to measure because they build on libraries of lemmas that have been accumulating since 1995. The correspondence result between the Motorola DSP microarchitecture and its microcode engine involved intermediate formulas that, when printed, consumed 25 megabytes (approximately 5000 pages of densely packed text) *per formula.* And the proof involved hundreds of such formulas. The correctness argument for one particular DSP microcode program required an extremely subtle generalization that took many days for the author to craft. The formal model of the Connected Limited Device Configuration (CLDC) JVM and bytecode verifier is almost 700 pages of densely packed text. The proof that a simple Java class, which spawns an unbounded number of threads, produces a monotonic increase in the value of a certain shared counter produces about 19,000 subgoals and requires about 84 megabytes to print.

In these senses, the theorems in which we are interested are little (but hard) theorems about big systems, or put another way, they are valuable and non-trivial theorems about parts of very complicated systems.

3 About ACL2

The ACL2 programming language is essentially the side-effect free (or "functional") part of ANSI Standard Common Lisp [14]. Here is the definition of a function `sum` that sums the integers from `n` down to `0`. The function might be defined in ACL2:

```
(defun sum (n)
  (if (zp n)
      0
      (+ n (sum (- n 1)))))
```

Thus, for example `(sum 6)` is `6 + 5 + 4 + 3 + 2 + 1 +0` or `21`.

The ACL2 theorem prover can prove that `(sum` n`)` is equal to

$$\frac{n \times (n+1)}{2}$$

when n is a natural number. In ACL2, this theorem is written this way:

```
(implies (natp n)
         (equal (sum n)
                (* n (+ n 1) 1/2)))
```

This theorem can be proved, using the axioms of propositional calculus, equality, arithmetic, and the recursive definition of `sum`.

The ACL2 theorem prover is a computer program that takes formulas like that one above as input and tries to find mathematical proofs. It uses rewriting, decision procedures, mathematical induction and many other proof techniques to prove theorems in a first-order mathematical theory of recursively defined functions and inductively constructed objects [5].

The ACL2 theorem prover can prove the formula above about `(sum n)`, completely automatically by induction on `n`, *provided* the user first instructs ACL2 to load the standard arithmetic lemma library.

The logic in which ACL2 theorems are stated is undecidable: no algorithm can be written that will determine whether or not a given formula is a theorem. ACL2's theorem prover is sound but incomplete: if it says "yes" the formula is a theorem; if it says "no" (or runs "forever") the formula may or may not be a theorem.

The theorem prover frequently needs help finding proofs. This help is provided by the user, in the form of key lemmas to prove first. As noted above, to prove the AMD K5 processor's floating-point division operation compliant with the IEEE standard, we needed to prove about 1,200 lemmas. Those lemmas formed the basis of a floating point arithmetic lemma library.

Many such lemma collections – called *books* in ACL2 parlance – are distributed with ACL2. Multiple books can be included into a single session to configure ACL2's database. Books are available for many different mathematical domains, including various parts of arithemtic (integer, rational, floating point), finite set theory, vectors, lists, etc.

The ACL2 home page [6] contains the ACL2 sources, images for many different platforms, installation instructions, books developed by the ACL2 users around the world, and documentation and papers about ACL2. ACL2 is distributed without fee under the terms of the GNU General Public License.

To learn to use ACL2 we recommend that you first read "A Gentle Introduction to ACL2 Programming" http://www.cs.utexas.edu/users/moore/publications/gentle-intro-to-acl2-programming.html, and then "ACL2 Programming Exercises 1" otherwise known as "Getting Started with ACL2," http://www.cs.utexas.edu/users/moore/publications/acl2-programming-exercises1.html. If you are still interested in ACL2, we recommend learning a bit about the theorem prover, by reading and doing the exercises in "How to Prove Theorems Formally," http://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/index.html. If you're still interested, buy [5] and do the exercises in it. It is not necessary to purchase [5] from Kluwer, which controls the hardback copyrights but not the paperback rights. For instructions for purchasing a spiral-bound paperback version of [5] (approximately at cost plus postage), see http://www.cs.utexas.edu/users/moore/publications/acl2-books/car/index.html. The solutions to all the exercises mentioned above are on the web.

Finally, the ACL2 community has a very active email help list to which all new users should subscribe. Many volunteers from this list answer questions and novices, especially, are encouraged to post questions. See the Useful Addresses link on the ACL2 home page.

References

1. B. Brock and W. A. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods*. Springer-Verlag, 1999.
2. 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.gz`.
3. David Greve and M. Wilding. A separation kernel formal security policy, 2003.
4. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.
5. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
6. M. Kaufmann and J S. Moore. The ACL2 home page. In `http://www.cs.-utexas.edu/users/moore/acl2/`. Dept. of Computer Sciences, University of Texas at Austin, 2004.
7. H. Liu and J S. Moore. Executable JVM model for analytical reasoning: A study. In *Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME '03)*, San Diego, CA, June 2003. ACM SIGPLAN.
8. W. McCune and O. Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 265–282, Boston, MA., 2000. Kluwer Academic Press.
9. J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. http://www.cs.utexas.edu/users/moore/-publications/marktoberdorf-03.
10. 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.
11. 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.
12. D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA., 2000. Kluwer Academic Press.
13. J. Sawada. Formal verification of divide and square root algorithms using series calculation. In *Proceedings of the ACL2 Workshop, 2002*. `http://www.cs.utexas.-edu/users/moore/acl2/workshop-2002`, Grenoble, April 2002.
14. G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.
15. 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.