Computer-Aided Reasoning with ACL2

J Strother Moore*

June 16, 2000

Abstract

I outline a one semester course on formal methods based on the ACL2 logic and theorem prover.

1 Justification

In this audience I do not have to justify the need to teach formal methods. However, I might have to justify my decision to teach just one tool: ACL2. "ACL2" stands for "A Computational Logic for Applicative Common Lisp." It is a functional programming language based on Common Lisp, a first-order mathematical logic with recursive definitions and inductive proofs, and a mechanical theorem proving system in the Boyer-Moore tradition. Along with Matt Kaufmann, I am a co-author of ACL2.

Why teach just one tool? Why ACL2 among all the choices?

Since I am a co-author of ACL2, the explanation is obvious at one level: ACL2 is the tool I know best. This is not said glibly. Enthusiastic teachers who deeply understand the subject matter tend to be good teachers. However, whenever one teaches a course based on a particular tool, it is incumbent upon the teacher to explore the tool's inadequacies, especially those that result from fundamental design decisions. Suggesting that all inadequacies can be hacked around or patched is a disservice. Some are the inevitable consequences of basic decisions and can only be remedied by building a different tool (with its own new set of inadequacies). Opening the student's eyes to this fact is important.

The argument for teaching just one tool is simple: a semester is not very long. If I were teaching a course on programming, I would rather the students learn one "first language" than several. Like programming, computer-aided reasoning is a jarring paradigm shift for most people. Once they have made the shift, it is easy for them to explore alternatives.

ACL2 is a good choice for the following reasons.

- ACL2 is based on a classic programming language, Lisp. The introduction to the logic is just an introduction to functional programming in Common Lisp.

*Department of Computer Sciences, University of Texas, Austin, TX 78712, moore@cs.utexas.edu.

- There is now a textbook introduction to ACL2, with problems and solutions: *Computer-Aided Reasoning: An Approach*, by Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, Kluwer Academic Publishers, 2000.
- The tool is free and runs on many platforms.
- The tool is rugged, well-documented online, and widely used.
- Within the ACL2 setting there is a natural way to study some other tools: a tautology checker, a model checker, a symbolic trajectory evaluator, and a first-order logic proof checker are implemented and verified in ACL2. These and other applications are presented in a companion volume to the book above. The volume is called *Computer-Aided Reasoning: ACL2 Case Studies*, (Kaufmann, Manolios, and Moore, eds.), Kluwer Academic Publishers, 2000. The volume presents material from 21 contributors.
- ACL2 users have formalized the semantics of the Java Virtual Machine, a Lisp compiler, the code-generation part of a compiler used by Union Switch and Signal to compile train-borne control programs, the semantics of several hardware description languages, a pipelined microprocessor architecture, a floating-point multiplier described at the register transfer level, and many other applications in hardware and software. Properties of these models have been proved and are available. Thus, selected applications can be studied in detail. I do not expect students to do such applications in this course, but I expect they can read and understand the specifications of one or two of these.
- Finally, and very importantly, ACL2 is not a pedagogical toy but an industrial-strength theorem prover used by such companies as Advanced Micro Devices, IBM, and Rockwell Collins Avionics to prove theorems about commercial hardware and software. For example, all of the elementary floating-point arithmetic hardware on the AMD Athlon microprocessor is IEEE compliant. This was proved using ACL2. In the process, bugs were found and fixed in designs that had survived hundreds of millions of tests.

Interested readers should visit the ACL2 home page at `http://www.cs-.utexas.edu/users/moore/acl2`. There links can be found to the books, solutions to exercises, case studies, the source code, installation instructions, several megabytes of hypertext documentation, and papers about ACL2 and its applications. The ACL2 home page also contains a link to the two books mentioned above, where one can find tables of contents, excerpts, solutions, etc.

2 Course Outline

Here is what I intend to teach in a fourteen week course at the University of Texas at Austin this Fall. This course will be taught to a small number of graduate students as a way of debugging my ideas for a subsequent undergraduate course. A laptop and LCD projector will be used in about two-thirds of the lectures to demonstrate the ideas. Students will be expected to do homework and mini-projects in interaction with the tool. Each item below denotes one week's content. The following textbook is required:

> *Computer-Aided Reasoning: An Approach*, Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, Kluwer Academic Publishers, 2000.

The table of contents of this book is available at `http://www.cs.utexas.edu/-users/moore/publications/acl2-books/car/index.html`. The readings below cover the main part of the textbook but not the appendices. The lectures at the end of semester are based on the ACL2 case studies book, which the instructor should have but which students would not need for this course.

The downside to this choice of text is that the book is expensive: $120. I understand that discounts (perhaps 20 percent) are available if the book is adopted for a class of 6 or more.

1. Introduction to the textbook and ACL2 web site. Instructions for local use of the tool. Demo: I will demo the system, the web page, and the documentation on my laptop. Assignment: Read the first 55 pages of the textbook; this is not too much because the first 25 pages are overviews. The remaining 30 pages are Chapter 3: an introduction to functional programming in Lisp. Homework: Selected exercises from Chapter 3.

2. Lisp programming. Simple Lisp programs presented and explained, including `length`, `append`, `reverse`, `member`, `union`, `flatten`. Demo: These functions will be developed on the board, then entered into the system and executed, with the usual typos and errors introducing reason to discuss system interaction details. Reading: Chapter 4 (Programming Exercises). Homework: Do the lab exercise described on the ACL2 Hyper-Card (see ACL2 home page) and selected exercises from Chapter 4.

3. Lisp programming continued. Take the students through a simple model of a machine. The purpose is two-fold: to show them how we will use Lisp to model other systems and to show them how a "big" system is just a collection of "small" functions. I will use the machine described in the paper "Proving Theorems about Java-like Byte Code" on my web page. The ACL2 script is available there too. A side-effect of this choice is that students will learn an abstract view of the JVM. The week will consist of a walk through that machine model. Most of the Lisp is accessible to them now. Demo: After presenting the model on the board or with transparencies, I will demo it, mainly by executing JVM programs on the model. Reading: Chapter 5 (Macros). Homework: selected exercises from Chapter 4 and Chapter 5.

4. Introduction to the logic as a traditional formal system, with axioms and rules of inference. I will lecture on Chapter 6 (The Logic), emphasizing the definitional principle and induction. Demo: None - it is time to think. Reading: Chapter 6. Homework: The easier problems in Chapter 6. I am not interested in teaching them how to do formal proofs, only in their knowing what formal proofs are.

5. Proofs about Lisp functions. I will do many proofs in class, essentially those presented in Chapter 7. Demo: None. Reading: Chapter 7 (Proof Examples). Homework: selected exercises from Chapter 7. These call for "hand proofs" comparable to the ones I've done in class.

6. A more careful look at recursive definition and induction. On why termination is important in a logical setting. On the duality of recursion and induction. Inspection of the ACL2 ordinals and well-foundedness. Demo: None. Reading: Review of the relevant sections of Chapter 6. Exercises: Termination arguments for simple proposed function definitions.

7. Introduction to the mechanical theorem prover. The user-level model. Demo: The sample proof of presented in Chapter 8. Reading: Chapter 8 (The Mechanical Theorem Prover). Homework: Use the system to do the proofs already done by hand. Most will be automatic. The drill will involve firing up the system and interacting with it.

8. How to use the theorem prover. "The Method," by which the user discovers what the system needs to know. Demo: I will prove a few of the theorems discussed in Chapter 10 (Theorem Proving Examples), using The Method to discover what is needed. Reading: Chapter 9 (How to Use the Theorem Prover). Exercises: Do the other theorems of Chapter 10.

9. Assignment of some mini-projects, drawn largely from the problems in Chapters 10 and 11. These include sorting, permutations, finite set theory, tautology checking, compiling, and bit-vector arithmetic. Numerous other ideas inevitably come up. The trick is to keep the students focused on sufficiently simple challenges.

During the next month, students will work on their projects. At the end they will turn in proof scripts checked by ACL2.

In the meantime, I will lecture on selected applications from the companion volume. The full scripts for each case study are available on the web. The primary emphasis of the following lectures will be on how to specify and model the problem. The existence of mechanically checked proofs will be alleged and occasionally demonstrated. But the emphasis is on what to specify and what can be proved.

10. Path finding in directed graphs. (Students continue to work on their mini-projects)

11. A mu-calculus model checker. (Pointer to symbolic trajectory evaluation) (Students continue to work on their mini-projects)

12. Floating point arithmetic. (Students continue to work on their mini-projects)

13. Compiler verification and Trojan horses. (Students continue to work on their mini-projects)

14. Theorems about Java byte code programs. (Students continue to work on their mini-projects)

I would not give a final test but let the student's homework and the ACL2-checked proof scripts for their projects determine the grade.

3 Student Profile and Follow Up

Prerequisites for the course above include the usual mathematical and programming background of an upper division computer science major, plus

- a course in mathematical logic and/or set theory, e.g., UT's PHL 313K Logic, Sets and Functions.
- a course in functional programming (e.g., UT's CS 307) in Haskell, Lisp or Scheme programming.

For three years I have taught an upper division undergraduate elective entitled CS 378T A Formal Model of the Java Virtual Machine. This is taught in the Fall and most of the students are seniors due to graduate the following May or August. Judging from the students who take my CS 378T, the course described here is suitable.

If the proposed course were taught in the Fall, then a course like my CS 378T would be a perfect follow-on course. In CS 378T, we study a formal model of the Java Virtual Machine (JVM), written in ACL2. My current course has two drawbacks as it is currently structured. First, we focus only on JVM specification, not proof of properties. Second, we cannot go as far into the JVM as I would like. For example, I generally give only one lecture on the JVM byte code verifier and two lectures on the JVM thread model. These drawbacks are due to lack of time. If the JVM course were offered as a follow-on to the formal methods course sketched here, both could be remedied, to the advantage of the JVM course.

The two topics "slighted" in my JVM course – the byte code verifier and threads – are wonderful examples of topics in the JVM that can benefit from formal models. For example, how is the byte code verifier specified? What, exactly, is the property it guarantees? Can you express the property using our model of the JVM or do you need another model? What is that other model? Threading is even richer, especially if one considers how to specify JVM programs that use multiple threads or operate concurrently with other threads.

Proof techniques for dealing with such programs and their specifications are also worthy of discussion and investigation.

More generally, the formal methods course I have outlined could be followed by a course in which students use ACL2 to specify and prove properties of larger and more realistic systems than studied in the proposed course. The last five weeks of the course above, in which the students hear lectures on case studies, just scratch the surface of the material available on these topics. Because of ACL2's use in industry, a number of such detailed studies are available. For example, using the ACL2 case studies book, above, and resources on the ACL2 home page, one could easily spend a semester investigating several of the following topics:

- models of hardware description languages, including VHDL, IBM's DE, and AMD's RTL, the specification of designs in those languages, and proofs of properties of implementations;
- the IEEE floating point standard and the correctness proof for (a sanitized version of) AMD's Athlon floating-point multiplier;
- the use of formal models as simulation engines: executing formal models and using such models to replace traditional C-based simulators; the article in the case studies book was written by engineers at Rockwell-Collins Avionics; the dual use of formal models for both testing and verification will be, I believe, a major driver in the adoption of formal models by industry;
- safety-critical issues and their impact on a compiler used by Union Switch and Signal in train borne control systems;
- a formal investigation into Ken Thompson's observation that the compiler boot-strap test is insufficient to show the absence of Trojan Horses;
- the specification and correctness proofs for various algorithms and their implementations, including a model checker, a symbolic trajectory evaluator, path-finding in graphs, a first-order predicate calculus proof checker, Knuth's generalization of McCarthy's 91-function, and several other abstract mathematical areas.

Most of these case studies are sufficiently deep that a student might spend most of a semester mastering it. The case studies contain ACL2 exercises and the students could develop and test their skills by doing those exercises.

4 Conclusion

This course teaches mechanized formal methods via the ACL2 approach. The course

- introduces formal logic via functional programming,

- lifts mechanically checked proofs from the level of traditional formal logic to the level of "blackboard" proofs to which students are accustomed,
- shows students a tool that they can use after a few weeks to do moderately interesting proofs like sorting and tree manipulation, and
- demonstrates that in more experienced hands significant industrial problems can be tackled.