# Computer-Aided Reasoning for the Masses

Panagiotis Manolios

College of Computer and Information Science
Northeastern University
360 Huntington Ave., Boston MA 02115, USA
pete@ccs.neu.edu

**Abstract.** A good test of the usability of formal verification techniques and tools is whether they can be successfully integrated into the undergraduate curriculum. Widespread use of formal verification techniques at the undergraduate level will require that we convince our colleagues that such techniques benefit our students more than the material they replace. Widespread use of formal verification tools will not only require robust, mature, and usable tools, but will highlight the true remaining usability issues. At Northeastern University, we embarked down this path several years ago. We introduced a required freshman class that teaches students how to reason about the programs they write using ACL2s, the ACL2 Sedan. So far, over 200 undergraduate students have taken the class.

## 1 Formal Methods at the Undergraduate Level

The case for teaching formal methods at the undergraduate level is easy to make: formal methods provide the foundations that allow us to understand, reason about, and gain predictive power over computational processes and artifacts. If we want to train engineers who can build the computational systems our society is increasingly dependent so that they are robust, dependable, and secure, then we have to teach these engineers how to model and reason about computational systems using mathematical rigor and precision. This is inherently different from what students learn in courses on algorithms. There the emphasis is on introducing paradigms that are useful for designing computationally efficient algorithms, such as dynamic programming and the greedy method.

The next step is to figure out where such a class fits in at the undergraduate level. What are the prerequisites? Well, students have to have a solid foundation in programming. At Northeastern, they get that in their first semester, when they learn how to design programs using a Scheme-based functional programming language [2]. The earliest we can teach a course on formal methods is in the second semester of the undergraduate program, and that is what we do. While there are obvious disadvantages to teaching such a course that early, the advantages are greater: if we want to change how students think about programming and computer science, we need to instill these values in them early. Teaching formal methods during the first year, helps shape how they think about computer science as they navigate through the undergraduate program.

A more unpleasant question is what course to displace. The choice we made at Northeastern was to replace a course on logic with the course on formal methods. We still do teach many of the concepts previously taught, but now the emphasis is on using logic to reason about computation. The advantage is that this plays into the strengths of our undergraduates: they can program, they have very good intuitions and strong interests in programming. In contrast, they often have difficulty seeing the point of logic presented in a classic, abstract setting.

Regarding course content, our focus is on teaching students how to reason about programs. The narrative is simple: we remind them that when they design programs they write informal "contracts" that specify the intended domains for function inputs and outputs. They write tests that check whether their programs give the expected result on particular inputs. In this class, they learn how to formalize their contracts, and how to specify the intended behavior of their programs not only on individual inputs, but across an infinite number of inputs. They also learn how to prove that their conjectured contracts are in fact true. There is a strong emphasis on gathering requirements and stating specifications, because these skills can and will be used throughout their careers, no matter what they do.

One of the challenges we faced is that the material is new for almost everyone in the class. None of them were playing around with theorem provers or proving correctness by hand in high school! In order to have a smooth transition from what they do know to what they do not know, we start with propositional logic. In this setting, we introduce notions such as tautology, satisfiability, falsifiable, and unsatisfiable. We show that one can falsify a conjecture by providing a single counterexample and that the lack of a counterexample implies that the conjecture is true. In fact, one can exhaustively test propositional conjectures, and they write their own validity solvers to do that. When we get to reasoning about programs, the story is almost the same: one can falsify a conjecture with a single counterexample. They understand this because they know how to evaluate programs. They also understand that if there is no counterexample, then the conjecture is true. What they do not know how to do is to check for this, as exhaustive testing does not work. So, we tell them that this is the power of logic: with finite work you can deduce an infinite number of conclusions, namely that exhaustive testing will not find a counterexample. The key proof technique used is induction. It is introduced as an extension of what they already known. When they design programs, they are (for the most part) data-driven, *i.e.*, the recursion scheme they use to define functions is based on the data definitions of the inputs to said functions. For example, in the recursive call for a function operating on lists, they can assume that the function will return the right result on the rest of the list, so all there is to think about is how to take this correct result and the first element of the list and put them together in a way that satisfies the contract for the function. Similarly, the data definition gives rise to an induction scheme that is used to reason about their programs. In the induction step, they can assume that their conjecture holds for the rest of the list, and using that, they have to show it holds for the whole list.

Another important aspect of our class is that we use lots of examples from computer science to motivate what we are doing. This has the benefit of introducing students to parts of computer science that they will not see until later on and teaches them to separate correctness aspects from efficiency aspects. For example, we discuss how to build circuits that perform bit-vector arithmetic, and how to specify and reason about what they do. We discuss various sorting algorithms. Simple sorting algorithms, such as insertion sort, are easier to verify than more efficient algorithms, such as quicksort, but they both have the same specification. We discuss compilation and design a simple compiler that given expressions generates instructions for a stack-based machine. We then specify what it means for the compiler to be correct, which is subtle, and prove it. We also prove program equivalence between programs whose running times are very different, a recurring theme in computer science.

## 2 Usability of Formal Verification Tools

A formal verification tool that is appropriate for the above class is one that is based on a function programming language and which is robust, well engineered, and usable. We use the ACL2 Sedan theorem prover (ACL2s). ACL2s is an Eclipse plug-in that provides a modern integrated development environment, supports several modes of interaction, provides a powerful termination analysis engine, and includes fully automatic bug-finding methods based on a synergistic combination of theorem proving and random testing. ACL2s is freely available, open-source, and well supported [1]. Installation is simple, *e.g.*, we provide prepackaged images for Mac, Linux, and Windows platforms.

ACL2s uses ACL2 as its core reasoning engine. ACL2 is a powerful system for integrated modeling, simulation, and theorem proving that is based on a simple Lisp-like applicative programming language [4, 3, 5]. Think of ACL2 as a finely-tuned racecar. In the hands of experts, it has been used to prove some of the most the complex theorems ever proved about commercially designed systems. Novices, however, tend to have a different experience: they crash and burn. Our motivation in developing ACL2s, the ACL2 Sedan, was to bring computer-aided reasoning to the masses by developing a user-friendly system that retained the power of ACL2, but made it possible for new users to quickly, easily learn how to develop and reason about programs.

Usability is one of the major factors contributing to ACL2's steep learning curve. To address the usability problem, ACL2s provides a modern graphical integrated development environment. It is an Eclipse plug-in that includes syntax highlighting, character pair matching, input command demarcation and classification, automatic indentation, auto-completion, a powerful undo facility, various script management capabilities, a clickable proof-tree viewer, clickable icons and keybindings for common actions, tracing support, support for graphics development, and a collection of session modes ranging from beginner modes to advanced user modes. ACL2s also provides GUI support for the "method," an approach to developing programs and theorems advocated in the ACL2 book [4].

The other major challenge new users are confronted with is formal reasoning. A major advantage of ACL2 is that it is based on a simple applicative programming language, which is easy to teach. What students find more challenging is the ACL2 logic. The first issue they confront is that functions must be shown to terminate. Termination is used to both guarantee soundness and to introduce induction schemes. Unfortunately, students quickly see functions that ACL2 cannot prove terminating, without user assistance. This involves reasoning about infinite ordinal numbers. Introducing the ordinal numbers and measure functions to freshmen is probably not a good idea, so we developed and implemented Calling-Context Graph termination analysis (CCG), which is able to automatically prove termination of the kinds of functions arising in undergraduate classes [6]. This definitely helped, but then we noticed that beginners sometimes define non-terminating functions, and it would be nice if we could provide termination counterexamples. We added this to ACL2s, and more [7].

Once their function definitions are admitted, new users next learn how to reason about such functions, which first requires learning how to specify properties. We have seen that beginners often make specification errors. ACL2s provides a new lightweight and fully automatic synergistic integration of testing and theorem proving that often generates counterexamples to false conjectures. The counterexamples allow users to quickly fix specification errors and to learn the valuable skill of generating correct specifications. This works well pedagogically because students know how to program, so they understand evaluation. Invalidating a conjecture simply involves finding inputs for which their conjecture evaluates to false. This is similar to the unit testing they do when they develop programs, except that it is automated.

The next area to focus on is better support for understanding rewrite rules, for driving the theorem prover, and for understanding what went wrong when the theorem prover fails.

## 3 Conclusions

Formal verification has been successfully used in industry by highly trained engineers, and is here to stay. In order for this technology to be more widely used, we have to train the next generation of engineers by integrating it into the undergraduate curriculum. This paper gave an overview of our experience in introducing formal methods at the freshman level at Northeastern University, where we used ACL2s to teach eight sections of a required second-semester freshman course entitled "Logic and Computation." The goal of the class is to teach fundamental techniques for describing and reasoning about computation. Students learn that they can gain predictive power over the programs they write by using logic and automated theorem proving. They learn to use ACL2s to model systems, to specify correctness, to validate their designs using lightweight methods, and to ultimately prove theorems that are mechanically checked. Students reason about data structures, circuits, and algorithms; they prove that a simple compiler is correct; they prove equivalence between various programs;

they show that library routines are observationally equivalent; and they develop and reason about video games.

It is much too early to declare this a success. What we need are good books and course materials in addition to usability improvements so that other faculty at different universities can try the same experiment. Success will depend on getting Formal Methods integrated into the undergraduate curriculum across many universities.

## Acknowledgments

Harsh Raju Chamarthi, Peter C. Dillinger, and Daron Vroon have made significant contributions to the development of ACL2s. We owe a huge debt to Matt Kaufmann and J Moore, the authors of ACL2. Frankly, most of what ACL2s does is to call ACL2. Matt and J have been very supportive of our efforts and have added significant functionality to ACL2 that supports ACL2s. Finally, Matthias Felleisen was a driving force behind the introduction of Logic and Computation as a freshman course at Northeastern University.

## References

1. Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. ACL2 Sedan homepage. See URL `http://acl2s.ccs.neu.edu/`.
2. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing.* MIT Press, Cambridge, MA, USA, 2001.
3. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, June 2000.
4. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, July 2000.
5. Matt Kaufmann and J Strother Moore. ACL2 homepage. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.
6. Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *Computer Aided Verification, CAV*, volume 4144 of *LNCS*, pages 401–414. Springer, 2006.
7. Panagiotis Manolios and Daron Vroon. Interactive termination proofs using termination cores. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2010.