Computational Logic in the Undergraduate Curriculum

Rex Page
University of Oklahoma
School of Computer Science
Norman, OK 73019 USA
+1 450 325 4042
page@ou.edu

ABSTRACT
Logic provides the mathematical basis for hardware design and software development. In fact, digital circuits and computer programs are logic formulas expressed in a formal language. Accordingly, educated computer scientists should have experience in reasoning about the formulas that their digital circuits and programs represent. An exemplary way to get this experience is to use computational logic in support of such reasoning. This paper searches the typical undergraduate curriculum in computer science for opportunities to include material on computational logic in the context of hardware and software design and implementation. It explains how computational logic has been included as an element of two courses required in most computer science programs. It discusses some successes and a few missteps that the author has experienced over the past nine years in developing this material and using it in the classroom, and it suggests opportunities for similar efforts in other courses.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/program verification – *correctness proofs, formal methods.*

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education, curriculum.*

General Terms
Design, Reliability, Languages, Verification.

Keywords
Formal methods, theorem provers, ACL2, software engineering, computer science curriculum.

1. RATIONALE
Logic provides the mathematical basis for hardware design and software development. In fact, digital circuits and computer programs are logic formulas expressed in a formal language. Accordingly, educated computer scientists should have experience in reasoning about the formulas that their digital circuits and programs represent. An exemplary way to get this experience is to use computational logic in support of such reasoning.

Using ACL2 [10] as a computational logic engine for exposing students to this technology provides some advantages over other choices. It employs the widely used syntax of Common Lisp and smoothly integrates its theorem proving system within this syntax. This is important because it alleviates problems that come with introducing radically new elements into the curriculum. One such problem is resistance from students, computer science faculties, and outside advisors of academic programs. A programming notation in widespread use over a long period is easier to sell than one that has seen less use. Practical applications of ACL2 in industry and government are important in this regard.

Another problem has to do with the steepness of the learning curve. Many computational logic systems require a substantial investment of intense study before they begin to pay off. ACL2 presents a relatively simple basis for stating properties of software (and of hardware models), one that most students recognize from their knowledge of predicate calculus. Furthermore, the reasoning engine of ACL2 automatically generates inductive proofs of many correctness properties. This makes it possible for students to succeed early. Once they understand how to state properties as formulas in logic, ACL2 delivers proofs of many of those properties without assistance.

The following sections describe experience with computational logic in three courses required of all computer science majors at the University of Oklahoma, discuss the impact of particular choices in material and projects, survey typical curricula for other opportunities to include similar material, and point out related efforts to put computational logic in baccalaureate courses. Some of the material in this paper has appeared in other forms in previous publications [13][14][15][16].[1]

2. SOFTWARE ENGINEERING
In my experience over the past five years of using ACL2 in software engineering courses, I have found that almost all

[1] This work has been supported by two grants from the National Science Foundation (http://www.nsf.gov/). The work in the area of software engineering was done in collaboration with Matthias Felleisen and some of his students at Northeastern University, NSF Grant 0633664. The discrete math/logic work was supported by NSF Grant 0082849.

students adapt to the programming methods that ACL2 requires within the first few weeks of the course, and that they learn to state software properties as part of that experience. Grumbling is common, but no more so than with any new computer system, and the high success rate dampens this reaction as the course progresses. There have been a few hard cases (students who grumble loudly and never stop), but they comprise less than 2% of over 200 students who have used ACL2 as their programming environment over the past five years in my software engineering courses.

The high level of acceptance has surprised me. It may be that students embrace the idea because it is one of the few things they learn in software courses that they didn't know in high school.

An even greater surprise has been the support of the advisory board of the School of Computer Science at OU. Most of them have reacted positively to the idea of focusing on correctness in software development and on using a theorem proving engine as an aid in this process. A contingent from the board attends final presentations by student teams of their software product at the end of the second semester of software engineering. Many of them comment positively on the use of ACL2 in the course. For example, Stephen Mercer, a lead developer on the LabVIEW team for National Instruments, observed that "one group showed the true power of this concept by writing a 3D rendering engine and then proving that their engine had no bit plane errors."

The material on computational logic in the two-semester, required, fourth-year course focuses primarily on correctness properties for functions and comprises about a third of the course content. The remaining content covers software design, architecture, and processes. First semester software development projects range from 50-line to 300-line individual exercises (five to seven projects) to thousand-line team projects (one or two projects). Project descriptions provide informal specifications for programs and properties the programs are expected to satisfy. Students formalize these specs as ACL2 code, theorems, and tests.

In the second semester of the software engineering course, student teams develop a software product over the course of the semester. There are a dozen or more separate deliverables spread more-or-less uniformly across the semester, culminating in a documented software product, usually comprising 2,000 to 6,000 lines of ACL2, including theorems and theorem-based test specifications. Project details can be found in [14] and some of its references.

For the future, I would like to design a multi-year series of projects with software security as a common theme. My intent is that the projects would lead to a body of software security components that continues to grow over the years. John Allen conjectures that security could be a killer application stimulating widespread use of formal methods in software development [1]. If so, this series of projects could tap into that opportunity.

Students learn to state correctness properties by generalizing tests for correctness that they would design as a standard part of a software development effort. For example, while the associativity of the append operation for lists is an important one in some contexts, the basic correctness of the operator can be stated as an equality between the first *n* elements of its result and its first argument, and an equality between its second argument and the elements of its result, excluding the first *n* of them (where *n* is the number of elements in the first argument). This is the kind of property that students verify using the ACL2 theorem prover.

Early on, students used ACL2 directly. They now use it from the Dracula programming environment [15][16] in the DrScheme system [8]. This environment provides the type of point-and-click interface that most students are accustomed to and extends ACL2 i/o to include graphics and interactions with keyboard and pointing devices and includes an automated testing facility that generates random tests based on statements of theorems [15]. In addition, a modern module facility [6], important for software engineering, is under development. A prototype is now in use.

The trickiest part of developing course material lies in designing projects. Students must define functions in a form that facilitates reasoning. Fortunately, the design patterns in the text of Felleisen, *et al*, *How to Design Programs* [7], provide good examples to follow. Students need a few tips on idioms that ACL2 handles well, such as counting down to zero when termination is triggered by a counter, rather then counting between general limits in either direction, but with this kind of instruction almost all students succeed in developing code to meet project requirements.

The greatest care must be taken in designing projects in which ACL2 will succeed in verifying the properties students specify for the functions they define. In early projects, ACL2 must succeed on its own, without hints or lemmas. In later projects, students can tolerate projects in which ACL2 needs help.

Students need guidance in choosing correctness properties to verify. Early projects describe properties in considerable detail. As the course progresses, detail declines but never disappears. I have not found a way to teach the material that enables most students to specify correctness properties entirely on their own.

Students acquire varying levels of ability in specifying useful properties, of course, and the abilities of the top quartile are entirely satisfactory. The bottom ten percent fail to grasp these concepts, and the rest fall somewhere in between.

The success rate in using the theorem prover to verify properties has a similar distribution. About five percent of the students acquire a facility with the theorem prover well beyond my expectations, and most students acquire a basic understanding that, at least, would provide a basis for learning to use a theorem prover effectively if they found themselves in a software development organization that encouraged or required it. Resources for getting started with ACL2 in this context are available via internet.[2] I would, of course, welcome collaboration with instructors who want to use or expand these resources.

I conclude that the primary obstacle to success in the introduction of computational logic in software development courses is the will to do it on the part of the computer science faculty.

3. DISCRETE MATHEMATICS / LOGIC

Prior to beginning the work on software engineering education described in Section 2, I spent four years developing material for the logic portion of a standard discrete mathematics course required of all baccalaureate students in computer science. The goal of this material was to apply the principles of logic and

[2] http://www.cs.ou.edu/~rlpage/SEcollab/tsc/tscsched.html

mathematical induction covered in most such courses directly to hardware and software artifacts, rather than using the contrived examples one sees in most discrete mathematics texts. While the material does not employ mechanized logic, it does serve to introduce students to stating and verifying properties of software and circuits through rigorous, mathematical reasoning.

During the period of development of this material, OU offered two sections of discrete mathematics per semester. This provided an opportunity to measure differences in the effects of the standard approach to the subject and an approach applying logic to the task of verifying properties of digital circuits and software components. Data gathered over a period of three years on the performance of students from the two versions of discrete math in a subsequent software development course showed a statistically significant difference favoring the applied logic approach [13].

The primary content of the logic component of the discrete math course, which comprised about two thirds of the course material, focused on reasoning about digital circuit designs and software components expressed in an equation-based notation, namely the programming language Haskell, chosen because its syntax closely resembles standard algebraic notation. Other formal languages can be equally effective. For example, the TeachLogic Project, which covers similar material, employs Scheme.[3]

Students learn to use logic and induction to verify properties of computing artifacts. They exhibit their abilities by solving about a hundred homework exercises and a couple dozen exam questions, about 90% of which require mathematical proofs.

Proofs are carried out by hand, rather than by way of a computational logic. Students use an automated proof checker in the propositional logic portion of the course, and they see a demonstration in which the ACL2 theorem prover succeeds (in microseconds) in proving some of the properties the students have been recently struggling with. This leaves a positive impression and motivates students for ACL2 usage in software engineering.

After completion of the project comparing the effects of the standard approach to logic in discrete math and the applied logic approach, the OU faculty decided to expand logic coverage by splitting discrete math into two courses, one in which all of the subject matter is logic applied to verification of hardware and software properties, and another covering the remaining discrete math topics (combinatorics, relations, functions, computational complexity, recurrences, and graph theory).

The new course goes by the name “Applied Logic for Hardware and Software” and acts as the prerequisite for both discrete math and computer organization. Properties verified in lectures include the usual properties of the list concatenation operation (associativity and the proper relationships between operands and result), several properties of merge sort (ordering, element and length preservation, computation time), correctness and time consumption of the Russian peasant algorithm, full correctness for a formal model of a combinational circuit for a ripple-carry adder, and dozens of other useful computational components.

Students apply the ideas in homework and exam problems similar to examples covered in lectures. In the end students have seen logic in action in a hardware design and software development. One benefit is that students are well motivated to study logic because they see its relevance to computer science. Another is that students can apply logic in subsequent course work.

Early in the course, function definitions appear as ordinary algebraic equations expressing simple properties of operations on data structures. I do not point out until later that these simple properties actually define functions. Instead, I focus on using logic, equation-based reasoning, and induction to derive more complex properties from the simple ones expressed in the equations (which happen to be definitional).

Later, students discover that the equations do, in fact, define functions and that collections of functions comprise software specifying complex computations. The course is not about functional programming. It is not about programming at all. Students learn just enough about Haskell to manipulate formulas in reasoning about properties of engineering artifacts. It is in subsequent courses, such as software engineering, where students apply these ideas to their own software designs.

As with the introduction of computational logic in software engineering, the primary obstacle to the application of logic to engineering artifacts in the discrete math curriculum is the desire of the faculty to do use this approach. It requires careful preparation, but resources are available as a starting point [11].[4]

4. RELATED WORK

Evidence of the use of computational logic in undergraduate computer science courses is hard to find. Manolios uses ACL2, via the ACL2 Sedan [4], in a lower division logic course introduced experimentally by Felleisen [5]. Tinelli[5] has included assignments in the use of KeY [2], a tool for theorems and proofs about programs written in a subset of Java. Courses employing Haskell often use QuickCheck [3], which gives students practice in stating properties as logic formulas, an important skill in using computational logic systems. Jackson´s Alloy system is used in undergraduate classes and exposes students to logic as a tool for stating and verifying properties of software components [9].

5. CURRICULUM OPPORTUNITIES

What courses provide the best opportunities to introduce computational logic to undergraduate computer science students? Designing a new course specifically for that purpose or choosing an existing course in formal methods is a straightforward path, and probably the easiest path to follow because the faculty seldom puts obstacles in the way of new courses or reasonable changes in the content of elective courses.

However, if knowledge of computation logic is important for computer scientists, it must occupy a place in the required core of the computer science curriculum. That is, computational logic must be integrated into required courses. One way to look for productive targets would be to focus on courses that almost all computer science programs require.

To get an idea of what those courses are, one could take a random sample of baccalaureate computing programs and make a list of

[3] http://www.teachlogic.org

[4] http://www.cs.ou.edu/~beseme/

[5] http://www.cs.uiowa.edu/~tinelli/classes/181/Spring08/

the most frequently offered courses. As an initial attempt in that direction, I chose ten computer science programs at random.[6]

Seven courses were required in 90% or more of the surveyed programs and no other courses were required in over 60% of the programs. The high-probability courses were programming i/ii, discrete math, computer organization, data structures, operating systems, and software engineering.

We have a start on introducing computational logic in discrete math and software engineering. Programming i/ii and data structures are difficult targets because most computer science faculties jealously guard the content of those courses [12]. That leaves computer organization and operating systems.

Computer organization is an inviting target because computational logic is widely used in practice to verify properties of digital circuits, which comprise a part of the content of most computer organization courses. Probably there would not be a lot of space for hands-on practice with computational logic in a computer organization course, but there might be space for some lecture material, a homework problem, and an exam question.

Operating systems are a tempting target because of the security issue. The idea of equation-based software for programming operating system components would be hard to sell, but ACL2 models of certain operating system functions might be feasible.

Therefore, I expect that the use of computational logic would be feasible in at least four courses required in almost all baccalaureate CS curricula without changing course descriptions (computer organization, operating systems, software engineering, discrete math). Of these, discrete math and software engineering are the easiest targets because they are rarely micromanaged by the faculty, and they have a lot of space for applications of logic.

The choice of a formal language for expressing predicates about software and hardware artifacts carries no special importance for intellectual content, but practicalities do matter. ACL2 has the advantage of fitting within the confines of a widely used and easy to learn syntax (Common Lisp). It also has an extremely powerful theorem prover, which flattens the learning curve and allows students to succeed early without overburdening them with details that fall outside normal course content.

It is fair to ask whether computational logic belongs in any course in the undergraduate curriculum, since its coverage must displace other topics. My assumption is that software and hardware implementation cannot reach acceptable standards of reliability without a sound basis in principles [1]. Since software and hardware designs are, literally, formulas in logic, logic provides an obvious body of principles suited to the purpose.

To be used effectively in designing software and hardware, logic must be checked through the last detail, and that is feasible only with mechanization. Fortunately, modern computational logics are up to the task, and at least one of them is a useful tool for some standard undergraduate courses.

[6] The “random” universities were those with Division 1A football titles in 2008: Virginia Tech (ACC), Oklahoma (Big 12), Cincinnati (Big East), Penn State (Big Ten), East Carolina (C-USA), SUNY Buffalo (MAC), Utah (Mtn West), USC (PAC 10), Troy (Sun Belt), Florida (SEC), Boise State (WAC). Troy’s website lacked CS curriculum information.

6. REFERENCES

[1] Allen, J. 2008. "Whither software engineering", Workshop on Phil. and Engr. (London, UK, Nov 10-12, 2008) 48-49 http://www.raeng.org.uk/policy/philosophy/pdf/abstract_papers.pdf

[2] Beckert, B., Hähnle, R., and Schmitt, P.H., Eds. 2007. Verification of Object-Oriented Software: the KeY approach. LNCS 4334. Springer-Verlag.

[3] Claessen, K. and Hughes, J. 2000. "QuickCheck: a lightweight tool for random testing of Haskell programs", Proc. of the 5th ACM SIGPLAN International Conference on Functional Programming (Montreal, Canada, Sep 18-21, 2000) 268-279.

[4] Dillinger, P.C., Manolios, P., Moore, J.S., and Vroon, D. 2006. "ACL2s: the ACL2 sedan", User Interfaces for Theorem Provers Workshop (Seattle, WA, August, 2006) In: Electronic Notes in Theoretical Computer Science, 174,2, 3-18. http://www.sciencedirect.com/science/journal/15710661

[5] Eastlund, C., Vaillancourt, D., and Felleisen, M. 2007. "ACL2 for freshman: first experiences", Proc. of the 7th International Workshop on the ACL2 Theorem Prover and Its Applications (Austin, TX, Nov. 15-16, 2007) 200-211.

[6] Eastlund, C. and Felleisen, M. 2009. "Toward a practical module system for ACL2", Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (Savannah, Georgia, Jan 19-20, 2009) 46-60.

[7] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. 2001. How to Design Programs. MIT Press.

[8] Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. 2002. "DrScheme: a programming environment for Scheme", J. of Functional Programming 12, 2 (Mar. 2002) 159-182.

[9] Jackson, D. 2006. Software Abstractions. MIT Press.

[10] Kaufmann, M., Manolios, P., and Moore, J. 2000. Computer Aided Reasoning: An Approach. Kluwer.

[11] O´Donnell, J., Hall, C., and Page, R. 2006. Discrete Mathematics Using a Computer, 2nd edition, Springer.

[12] Page, R. 2001. "Functional programming ... and where you can put it", ACM SIGPLAN Notices 36,9 (Sep. 2001) 19-24.

[13] Page, R. 2003. "Software is discrete mathematics", Proc. of the 8th ACM SIGPLAN International Conference on Functional Programming (Uppsala, Sweden, August 25-27, 2003) 79-86.

[14] Page, R. 2007. "Engineering software correctness", J. of Functional Programming 17, 6 (Nov. 2007) 675-686.

[15] Page, R., Eastlund, C., and Felleisen, M. 2008. "Functional programming and theorem proving for undergraduates: a progress report", Proc. of the Workshop on Functional and Declarative Programming in Education (Victoria, B.C., Canada, Sep. 21, 2008) 21-29.

[16] Vaillancourt, D., Page, R., and Felleisen, M. 2006. "ACL2 in DrScheme", Proc. of the 6th International Workshop on the ACL2 Theorem Prover and Its Applications (Seattle, WA, Aug. 15-16, 2006) 107-116.