Integrating Logic into the Computer Science Curriculum*

Ian Barland[1], Matthias Felleisen[1], Kathi Fisler[1,3], Phokion Kolaitis[2], and Moshe Y. Vardi[1]

1: Department of Computer Science, Rice University, Houston, TX, USA
2: Department of Computer Science, University of California at Santa Cruz, USA
3: Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA, USA

{ian,matthias,kfisler,vardi}@cs.rice.edu, kolaitis@cse.ucsc.edu

Abstract

Computer science programs prepare students to construct systems. System construction requires reasoning about the interactions, capabilities, and limitations of components. A good degree program should teach students the tools that assist in these tasks. Logic is the most fundamental such tool, as industry has begun to appreciate. Software developers analyze program behavior during design, debugging, and testing. Hardware designers perform minimization and equivalence checking on circuits. Operating system designers validate routing, scheduling, and synchronization protocols. Formal logic underlies all of these activities. A solid preparation in applied logic should therefore increase students' abilities to reason about complex systems.

Unfortunately, few undergraduate computer science curricula prepare students adequately in logic. The typical student sees a few weeks of truth tables and propositional logic in a discrete mathematics course. Such a cursory introduction does not illustrate how applying logic helps with practical work. Upper-level applications courses neither discuss the role of logic nor reiterate the logical concepts covered in earlier courses. Accordingly, students perceive logic as irrelevant to their careers. Departments must modify their curricula to rectify this situation.

We plan to develop a series of modules that seamlessly integrate logic and logical software tools into existing, widely taught computer science courses. Integration over several semesters melds theory and practice, gives students concrete examples of how logic serves as a valuable tool, and increases students' abilities to absorb the logical concepts by repeated reinforcement. Emphasizing software tools makes the logical applications concrete and appeals to students with diverse learning styles. Modules, complete with lecture notes, presentations, problem sets, and tools, would facilitate curricular treatment of applied logic at all levels of college education, particularly in departments with scarce resources.

Modern computing is undergoing a subtle revolution. The Web has turned our computers into bank tellers, shops, and various information devices. Computers support more critical functions, such as air traffic controls, patient supervision, and tax collection. These applications demand a higher degree of safety, security, and robustness from computer systems at all levels. Even popular press publications, such as the Economist, the New York Times, and Byte Magazine, have joined the growing choir of voices calling for increased system reliability in computing [6, 14, 19, 29].

Computer science as a discipline has already yielded several techniques for implementing safe, secure, and robust systems. Significant progress has been made in the development of mathematically based

*This position statement is adapted from a grant proposal to develop the materials discussed herein.

programming languages, techniques and tools for specifying, designing, and verifying such systems. Programming languages such as Java include well-known features for supporting safe and secure code development [16]. Designers have applied specification and analysis techniques to a wide range of projects, including avionics software systems, air-traffic control systems, nuclear reactor controllers, telephone switching systems, and NATO security policies [7]. Hardware designers at companies such as Intel and IBM develop state-of-the-art microprocessors using formal verification systems based on research in applied logic and algorithms [26, 32]. Program analysis techniques such as static debugging [37] and extended static checking [11] have been applied to corporate consumer products; requirements analyses have detected flaws in commercial word processors [22]. The resulting improvements in system quality attest to the utility of these techniques in real-world system construction [9, 10, 23].

Computer science education needs to catch up. Everyone involved in information technology, from designers to programmers to managers to researchers, must understand the mathematical and logical foundations of advanced system construction. Unfortunately, the typical computer science curriculum makes no systematic attempt to expose students to the relationship between these foundations and practical computer science.

Developing Safe, Secure, and Robust Systems Different applications require different degrees and forms of safety, security, and robustness. A Web administrator, for example, would like guarantees that CGI-scripts never cause core dumps. Electronic commerce software should guarantee that sensitive information is always encrypted when it enters a public network. Air-traffic control systems should raise warnings whenever aircraft are too close together. Identifying these requirements is a *domain-specific* skill. One cannot reasonably define the requirements on a banking system, for example, without knowing the kinds of transactions users need to perform and the level of sensitivity of the information used in each transaction. The skills of stating requirements and analyzing systems, however, are largely domain-independent. They involve expressing ideas in formal notations and establishing relationships between statements.

Logic provides the mathematical foundation for these activities [36]. Its role in reasoning about systems is evident in several situations:

- Software developers must analyze the behavior of their programs. Assertions about behavior are logical statements about the language semantics; program analyses derive information related to the assertions through logical rules [5, 13].
- Hardware designers use gates with a direct correlation to Boolean logic; circuit minimization and equivalence checking rely on the logical equivalence of designs [18].
- Network designers propose protocols for routing, scheduling, and synchronization. Database designers develop protocols for concurrency control and data locking. Establishing protocol correctness requires a specification language and corresponding verification framework [20].

In each case, logic serves as a tool for modeling systems and as a language for expressing their desired properties. It also provides the required analysis frameworks, either through automated decision procedures or through semi-automatic proofs based on standard techniques such as induction and equational derivations.

Logic's role in systematic design is not accidental. In addition to being the foundation of computer science, over the last twenty-five years logic has played an increasing role in application areas. The relational database model, which is grounded in first-order logic, revolutionized database design and remains the preeminent model today [8]. Programming language semantics rest on the λ-calculus and constructive logic [3]. The ideas of type-safety and garbage collection result from these studies and have finally entered mainstream computing. Other significant applications abound, as evidenced by research results. To date,

one-quarter of all ACM Turing Awards have been awarded for contributions related to logic in computer science [1]. In recognition of this impact, logic is often called "the calculus of computer science". Indeed, John McCarthy, winner of the 1971 Turing Award, said in 1960 that "it is reasonable to hope that the relationship between computing and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last."

Students and System Development Skills Developing robust systems requires three main skills: (a) the ability to identify the robustness requirements for an application; (b) the ability to state robustness requirements precisely; and (c) the ability to analyze a program relative to its robustness requirements. Most undergraduates lack these fundamental skills. The investigators have observed that students have difficulties formalizing statements in logical notations and in developing arguments in support of their claims. Many, if not most, can describe properties of their own programs only at the implementation level, not the specification level. Indeed, they often express resentment at being asked to defend the correctness or other properties of their own programs. Even if they appreciate the connections between reasoning about programs and good programming, they find that it is too difficult and too time-consuming to perform this reasoning.

The first observation reflects a long-standing and well-known problem. Almstrum's studies [2] show that computer-science students generally have a harder time understanding logical concepts than non-logical ones. The other three observations, however, are far more disturbing. Not only are students unable to reason about their programs, they do not even understand why doing so is important; furthermore, they have developed negative attitudes towards the whole process.

Still, the process is important for computer-based work. Dennis Frailey, a software engineering researcher at Texas Instruments, stressed logical reasoning as one of the crucial skills which most of today's computer science graduates lack [15]. One recent Rice University computer science graduate cited this as one of the two skills she most needed—*and failed to learn in college*—in her job as a software engineer. Other colleagues, both at other universities and at other companies, have raised similar complaints. The problem cannot be dismissed as a case of universities and industry disagreeing on the skills required of degree holders. Understanding why a program solves a particular problem is a fundamental skill, one which employers value in their employees. That students fail to learn or appreciate this fundamental skill suggests a problem with existing curricula.

Where Curricula Fall Short Two main problems plague current curricula:

1. The peripheral role of logic in current curricula fosters the impression that logic is irrelevant to students' careers. Those courses that students consider directly relevant to their careers discuss neither the potential applications of logic nor the logical concepts involved in reasoning about the material.

2. Instead, basic logical concepts are typically relegated to a few weeks of a discrete mathematics course, covering truth tables, propositional logic, and basic proof techniques. Since applications to realistic problems are beyond the students' domain knowledge, these courses often ignore them. Upper-division logic courses, offered at few universities and taken by fewer students, ignore applications and come too late in students' careers.[1]

Simply stated, the problem is a separation of theory and practice; few students appreciate the utility of theoretical material in isolation. Furthermore, two flaws in the traditional presentation of logic compound the problem:

[1] Alternately, some schools have students take logic through the philosophy or math department; while such classes may spend more time on some aspects of logic, they still do not emphasize computer science applications.

3. Courses present only limited views of logic. Students see logic mainly in the context of Boolean algebra or simple hardware components, which overlooks logic's other roles in specification languages and decision procedures. This reinforces the belief that logic is irrelevant to most software.

4. Courses present logic in a deductive, passive style that directly conflicts with most students' learning styles. Accordingly, few students internalize or appreciate the logic material.

Learning styles have a large impact on how students respond to course material [27]. Research on learning styles attempts to correlate common personality traits with success under particular styles of instruction, assignments, and assessment. Two students with vastly different learning styles can respond quite differently to the same material. Felder [12] describes learning styles along five axes: visual vs. verbal, inductive vs. deductive, active vs. reflective, sequential vs. global, and sensory vs. intuitive.

Conventional logic curricula target intuitive, verbal, deductive, reflective, and sequential learners. Surveys of engineering students suggest that this set of preferences accounts for roughly a third of male students and a fifth of female students [28]. While these figures serve only as a guideline, they strongly suggest that a wider variety of teaching methods are needed to encourage students to develop strong skills in applied logic. Similar results have already been seen in other disciplines; Tobias' review of successful undergraduate science curriculum reform efforts highlights several cases in which accommodation to different learning styles played an important role [34].

In conjunction, the four problems listed in this section result in few computer science undergradates with even the foundation, much less well-developed skills, needed to develop safe, secure, and robust systems. Addressing only a subset of the problems is insufficient. Improving students' reactions to logical techniques and their ability to use them in practical settings requires curricular enhancements that address all four problem areas simultaneously.

The Proposed Solution: We aim to integrate applied logical reasoning and logical software tools into existing, widely-taught computer science courses. This approach is well-suited to addressing the problems with existing curricula for several reasons:

1. Seamless integration should greatly increase students' motivation. It provides concrete examples of how logic serves as a valuable tool in practical work. The success of covering context-free grammars in compilers courses proves the utility of such mergers. Students who complain about the irrelevance of other material in theory of computation courses accept lectures on context-free grammars in the context of parser design.

2. Integration and the use of logical software tools help students who grasp concepts better in concrete, as opposed to abstract, learning situations. As students work with software tools to analyze their programs, they understand what theorems mean in a specific domain and how proofs verify relationships between claims.

3. Finally, integrating material into several courses reinforces logical concepts over the entire undergraduate curriculum, thus increasing students' ability to absorb the material.

We plan to develop ready-made teaching modules with guidelines on how to integrate them into standard courses (such as databases, artificial intelligence, software engineering, architecture, compilers, and concurrent programming). The modules, each providing roughly three weeks of material, would allow departments to integrate material on logical modeling and robust system development without introducing new courses. In most cases, these would modules seek to apply the perspective of logical modeling and robust specifications to *existing topics* in standard curricula. Including such modules in a class would require

displacing only minimal material in most cases. Providing a series of modules would increase the opportunities for departments to cover logical material across multiple courses in a uniform fashion; this goes beyond efforts reported at other institutions that integrate formal material into isolated courses [17, 25, 35]. Each module should consist of a short text book, on-line (HTML or PowerPoint) lecture notes, presentations, programming exercises, problem sets, and guidelines for using appropriate software tools.

Justification for the Proposed Solution: Several people, including both researchers and educators, have called for a wider integration of logical concepts into the CS curriculum [21, 30, 33]. A survey of computer science faculty indicates a general belief that more rigorous techniques are needed in undergraduate software development courses [31]. In addition, the investigators' experience supports the integration of logical software tools and the overall module approach. Rice University's sophomore-level course in discrete mathematics is now almost entirely tool-based, using combinations of Mathematica and a logical tool called Hyperproof [4] to reinforce mathematical concepts. Student response to the approach has been enthusiastic.

As a proof-of-concept, two of the authors, Professor Felleisen and Dr. Fisler, developed and taught a pilot Software Engineering module in the Fall of 1998. Students learned how to state formal properties about simple programs, and explored equational, inductive proofs for establishing those properties using a theorem prover called ACL2 [24]. The pilot gauged how well logic could be integrated into an applications course, both in terms of the material, and also from the students' perspective. Students' initial difficulties with the material underscored their problems in thinking about correctness criteria for their programs. Overall though, students' reaction to the material was extremely positive. Their evaluations cited the material as interesting, useful, and thought-provoking.

Conclusion: Computer Science curricula need to provide students with a solid foundation in the reasoning skills required to design and develop computational artifacts. Logic, the so-called "calculus of Computer Science", underlies most approaches to constructing safe, secure, and robust systems. Curricula should therefore develop students' abilities to use logic as a tool in applied settings. We believe that providing concrete modules of teaching materials – including lecture notes, assignment suggestions, and tool guidelines – would best facilitate departments' efforts to include this material in their curricula. We call on other researchers and educators to help develop and test these modules in their courses.

References

[1] ACM Turing Award lectures: The first twenty years. ACM Press, 1987.

[2] Almstrum, V. L. Student difficulties with mathematical logic. In *Proc. DIMACS workshop on Teaching Logic and Reasoning in an Illogical World*, July 1996.

[3] Barendregt, H. P. *The lambda calculus: its syntax and semantics*. North-Holland, 1984.

[4] Barwise, J. and J. Etchemendy. *Hyperproof*. CSLI Lecture Notes. University of Chicago Press, 1994.

[5] Bourdoncle, F. Abstract debugging of higher-order imperative languages. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 46–55, June 1993.

[6] Cairncross, F. Midnight's children (the millennium bug survey). *The Economist*, 348(8086):15–17, September 19 1998.

[7] Clarke, E. M., J. M. Wing et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[8] Codd, E. A relational model for large shared data banks. *Communications of the ACM*, 13:377–387, 1970.

[9] Craigen, D., S. Gerhart and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NIST GCR 93/626, U.S. National Institute of Standards and Technology, March 1993.

[10] Eiriksson, A. T. The formal design of 1M-gate ASICs. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, 1998.

[11] Extended static checking. `http://www.research.digital.com/SRC/esc/Esc.html`.

[12] Felder, R. Reaching the second tier: Learning and teaching styles in college science education. *Journal of College Science Teaching*, 23(5):286–290, 1993.

[13] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[14] Flaw in email programs points to an industrywide problem. The New York Times, July 30 1998.

[15] Frailey, D. Colloquium, Rice University, Department of Computer Science, 1996.

[16] Gosling, J., B. Joy and G. Steele. *The Java language specification*. Addison-Wesley, 1996.

[17] Gries, D. and F. B. Schneider. *A logical approach to discrete math*. Springer-Verlag, 1993.

[18] Hachtel, G. D. and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[19] Halfhill, T. R. Crash-proof computing. *Byte*, April 1998.

[20] Holzmann, G. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[21] J. Paul Myers, J. The central role of mathematical logic in computer science. *SIGCSE Bulletin*, 22:22–26, 1 1990.

[22] Jackson, D. and C. Damon. Elemnts of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[23] Jacky, J., J. Unger, M. Patrick, D. Reid and R. Risler. Experience with Z: Developing a control program for a radiation therapy machine. In *Proc. Z Users Meeting*, 1997.

[24] Kaufmann, M., P. Manolios and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[25] Kelley Sobel, A. E. Experience integrating a formal method into a software engineering course. In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pages 271–274, 1996.

[26] Kurshan, R. Formal verification in a commerical setting. In *Proc. 34th ACM/IEEE Design Automation Conference*, 1997.

[27] Lawrence, G. *People Types and Tiger Stripes*. Center for Applications of Psychological Type, Inc., third edition, 1993.

[28] McCaulley, M. H. The MTBI and individual pathways in engineering design. *Engineering Education*, pages 537–542, July/August 1990.

[29] Minasi, M. *The Software Conspiracy: Why Software Vendors Produce Faulty Products, How They Can Harm You, and What You Can Do About It*. McGraw Hill, 1999.

[30] Morgenstern, L. and R. H. Thomason. Teaching knowledge representation: Challenges and proposals. In *Proc. KR2000 Conference*, pages 725 – 733, 2000.

[31] Palmer, T. V. and J. C. Pleasant. Attitudes toward the teaching of formal methods of software development in the undergraduate computer science curriculum: a survey. *SIGCSE Bulletin*, 27(3):53–59, September 1995.

[32] Schlipf, T., T. Buechner, R. Fritz, M. Helms and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 41(4:5), 1997.

[33] The 21st century engineering consortium workshop , March 1998. `http://lal.cs.byu.edu/21cec/announ2.html`.

[34] Tobias, S. *Revitalizing Undergraduate Science: Why some things work and most don't*. Research Corporation, 1992.

[35] Tremblay, G. An undergraduate course in formal methods: "description is our business". In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pages 166–170, 1998.

[36] van Leeuwen, J., editor. *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publishers, 1990.

[37] Weise, D. Personal communication, Montreal, June 17 1998.