Computational Logic in the First Semester of Computer Science: An Experience Report

David M. Cerna[1], Martina Seidl[2], Wolfgang Schreiner[1], Wolfgang Windsteiger[2], and Armin Biere[1]

[1]Institute of Formal Methods and Verification, Johannes Kepler University
[2]Research Institute for Symbolic Computation, Johannes Kepler University

Abstract

Nowadays, logic plays an ever-increasing role in modern computer science, in theory as well as in practice. Logic forms the foundation of the symbolic branch of artificial intelligence and from an industrial perspective, logic-based verification technologies are crucial for major hardware and software companies to ensure the correctness of complex computing systems. The concepts of computational logic that are needed for such purposes are often avoided in early stages of computer science curricula. Instead, classical logic education mainly focuses on mathematical aspects of logic depriving students to see the practical relevance of this subject. In this paper we present our experiences with a novel design of a first-semester bachelor logic course attended by about 200 students. Our aim is to interlink both foundations and applications of logic within computer science. We report on our experiences and the feedback we got from the students through an extensive survey we performed at the end of the semester.

1 Introduction

Recently, J. A. Makowsky and A. Zamansky (Makowsky and Zamansky, 2017) reported on an undesirable, but ubiquitous fact: courses on logic are slowly disappearing from the computer science curriculum at many universities. This has nothing to do with logic being outdated. For example, at the very basic level of computing, there is a close relationship between circuits and Boolean formulas and at a higher abstraction level, Boolean formulas are core concepts in all modeling and programming languages. Because of its rich inference mechanisms, logical formalisms form the basis of the symbolic branch of artificial intelligence (Russell and Norvig, 2010) and the core of modern verification technology relies on automated reasoners that evaluate logical formulas. For decades, the desire and necessity for verified hardware and software has grown in major software and hardware companies (Kaivola et al., 2009; Calcagno et al., 2015; Cook, 2018) driving investments in advancing logical reasoning tools.

While logic remains fundamental and pervasive, its importance is not directly obvious in the way it is classically taught. The classical logic course covers syntax and semantics of various logical languages, proof systems and their properties for

proving or refuting logical formulas, and maybe some encoding of combinatorial problems like graph coloring. Exercises are often very abstract and solved with pen and paper. We experienced that from such exercises it is unclear to students what are the practical applications, and often logic is perceived as just another math subject. This teaching approach is in strong contrast to our research, where we develop efficient reasoning software for solving practical problems from AI and verification and where we use the theoretical concepts for proving that our approaches are indeed correct. Hence, it was a question for us to ask if we can integrate the practical, computational aspects of logic into our courses such that our students not only learn the basics of logic but also understand how to apply logic in application-oriented settings. Therefore, we completely redesigned our "Logic" course, a mandatory course in the first semester of the computer science bachelor of our university. For all kinds of logic we teach, we also give some software tools to the students allowing them to quickly gain some hands-on experiences. We have successfully applied this approach for the last five years. The informal feedback has so far been extremely positive. To capture the feedback in a more structured way, we developed a questionnaire which we distributed among the students participating in the most recent iteration of our course. In this paper, we report on the setup of our course, the questionnaire, and its outcome.

2 Related Work

As early computer systems became larger and more complex, awareness and costs of failures such as the Pentium-FDIV-Bug (Chen et al., 1996; Reid et al., 2016) raised. It is at this point, in the late 1980s and early 1990s when the necessity of verification technology and other logic-based methods was realized and their inclusion within computer science curricula was addressed. Also, during this period we find our early references to literature discussing the addition of logic to computer science curricula. An important work wrote around this time, "Logic for Computer Science" by Steve Reeves and Michael Clarke (Reeves and Clarke, 1990) highlights some important topics which are considered in our course. Interestingly, as covered in the preface of the second edition, this book was of high demand and a second edition was released 13 years later. However, this work still takes a more classical approach to the subject. The book "Mathematical Logic for Computer Science" by Mordechai Ben-Ari (Ben-Ari, 2012), first released in 1993, has gone through three editions, the last of which, released in 2012, discusses prevalent subjects such as SAT solving and verification techniques, topics discussed in module 1 and 2 of our course. Such topics are discussed in (Huth and Ryan, 2004) as well. The importance of fundamentally logical questions to computer science is ubiquitously stated in Literature. Even earlier work by Jean Gallier (Gallier, 1985) covers the relationship between the theorem-proving methods and topics associated with computer science. By far the most classical work on this subject, to the best of our knowledge, is "the science of computer programming" by David Gries (Gries, 1981) which approaches programming from a logical perspective. For more modern works considering a similar approach to Gries consider "Software Abstractions: Logic, Language, and Analysis" (Jackson, 2012). There have also been a few influential works from this early period discussing approaches to logic education at several universities, it's relation to software engineering, and adoption of logic within curricula (Barland et al., 2000; Vardi, 1998; Lethbridge, 2000; Page, 2003; Wing, 2000). Furthermore, the following works (Kaufmann et al., 2000a; Kaufmann et al., 2000b; Reinfelds, 1995; Goldson et al., 1993) have provided interesting case studies concerning particular ways of integrating logic within the computer science curriculum. Particularly related to our

approach is (Kaufmann et al., 2000a) focusing on the use of the ACL2 theorem prover within the classroom.

Other then these collections of essential topics from logic for computer science there have also been studies of how certain didactic tools can be used for logic and computer science education. While we do not directly discuss how particular tools ought to be used in the classroom setting, we see further integration and development of the discussed methodologies into future iterations of the course. For example "Signs for logic teaching" (Eysink, 2001) which discusses representations and visualizations for the transfer of knowledge with respect to logic. Metaphorical games used in our course, especially in module 1, can be considered as visual aids in the learning process. The use of "serious games" for computer science education has also been investigated as of recently (Edgington, 2010; Muratet et al., 2009; Lee et al., 2014). Many of these investigations focus on programming or are indirectly related to programming, however, serious games for logic education have also been considered (Hooper, 2017).

Concerning the central topic of this paper, a discussion of our attempt to keep logic in computer science, there is the influential paper by J. A. Makowsky and A. Zamansky "Keeping Logic in the Trivium of Computer Science: A Teaching Perspective" (Makowsky and Zamansky, 2017) which we mentioned already in the introduction. This is not the only work of these authors concerning the difficult task of keeping logic in computer science (Makowsky, 2015; Zamansky and Farchi, 2015; Zamansky and Zohar, 2016). The problem addressed in these cases is how to connect the problems faced by students to the underlying concepts of logic they are presenting.

Also related to our course design is the adaption of proof assistants like COQ (development team, 2019) to make them usable in a classroom setting (Böhne and Kreitz, 2017; Knobelsdorf et al., 2017). While these works focused on introducing and using one specific tool, we present a course setting where we integrate multiple automated reasoning tools for teaching basic concepts of computational logic. Related in a broader sense, is computer-based logic tutoring software like (Huertas, 2011; Ehle et al., 2017; Leach-Krouse, 2017) that support the training of students.

3 Logic in Action

In the computer science curriculum of our university, the course "Logic" is scheduled in the first semester of the bachelor. As a consequence, we are confronted with a very heterogeneous audience of about 200 students including students who had only a one-year computing course at their high school as well as students who attended a five-year technical high school with a special focus on digital engineering and design. By offering a mix of mandatory and optional exercises, we let the students decide themselves how much they want to go into the depth of the topics discussed in our course.

The course consists of 12 lectures and 12 exercise classes. Both lecture and exercise classes are taught by professors of computer science or mathematics. Each lecture is a 90 minutes presentation of new content. After a short break, there is a mini-test, followed by an exercise class of 45 minutes. In the exercise class, the practical exercises covering the content of the same day's lecture are discussed for putting the theory into practice. Only part of the exercise sheet is solved in class, while the rest is left for practicing at home. The solutions of these exercises are not graded and can be discussed in the online forum of the course that is moderated by the professors.

For organizational reasons, the course is split into three modules: (1) propositional logic, (2) first-order logic, and (3) satisfiability modulo theories (SMT). The

first module takes four weeks, the second module takes six weeks and the third module takes two weeks. Each week the students have to take the aforementioned mini-test of 15 minutes. The test is about the content that has been covered in the lecture and exercise class of the previous week. It is closed-book and there are multiple-choice questions as well as free-style assignments. The mini-tests are manually graded by the professors—usually within a day. No mini-test can be repeated or taken at a later point in time. Each mini-test is worth up to 5 points. For passing the course, at least two positive mini-tests ($\geq$ 2.5 points) have to be in the first module, three positive mini-tests have to be in the second module, and one positive mini-test has to be from the third module. Up to four mini-tests can be replaced by a so-called lab exercise. The lab exercises are a kind of homework in which the students have to use logical software tools or solve some programming tasks. In addition to these lab exercises, there are also the *weekly challenges*, small bonus exercises, which allow the students to earn an extra point for the mini-test of the current week. The weekly challenges typically involve some sort of logical software as well.

One of the main distinguishing features of our course is the early adoption of reasoning technology. With this, the students immediately get an impression of the practical opportunities offered by logic. In the following, we report on three showcases illustrating the tool-based reasoning tasks integrated into our course.

3.1 Playing with SAT & SMT

To get experience in applying solving technology on practical reasoning problems, we ask the students to solve games with the aid of a SAT solver. A popular assignment is the encoding of Sudoku puzzles. The task is to translate the rules of 4×4-Sudoku into

3		4	
			1
	2		

propositional logic. The rules are as follows: Given a 4×4 grid, each of the small fields must contain exactly one of the numbers $(1,2,3,4)$ such that (1) no number occurs twice in a row, (2) no number occurs twice in a column, and (3) no number occurs twice in the four 2×2 grid (see figure). We also give the students a Sudoku instance that is not completed yet. The challenge is now to decide if this given Sudoku instance does have a solution. Even for these small sized Sudokus, the question is hard to answer without tool support.

The straight-forward encoding of the Sudoko simply reuses the ideas of graph coloring that was extensively discussed in the lecture before. The fields are represented by the nodes of the graph. The fields that may not contain the same numbers are connected in the graph. Solving the Sudoku then boils down to the question if there is a coloring of the graph using four colors such that no connected nodes have the same color. The formula is of a size such that it still can be generated by hand. However, some students prefer to implement a small script that outputs the encoding. For solving the formula, we developed a front-end for recent SAT

solvers that uses an input format students find more natural than the standard input format DIMACS, which is a list of numbers.

The language of propositional logic only provides Boolean variables and connectives. Hence the propositional encoding reduces the Sudoku-solving problem to a graph coloring problem. Later in the course we introduce SMT (Barrett et al., 2009) (Satisfiability Modulo Theory) that extends propositional logic by data structures like arrays or other data types like integers. Now the students can reformulate their Sudoku encoding by exploiting these advanced language features, learning that with more expressive languages encodings often become easier. The reasoning itself becomes more involved at this stage or even infeasible (depending on the concepts that are included in a language extension) and therefore we mostly rely on tool support (SMT solvers) in this part, even though we do explain basic algorithmic aspects of SMT solving.

3.2 Automatic Checking with RISCAL

RISCAL (RISC Algorithm Language) is a language and associated software system for the formal modeling of mathematical theories and algorithms in first order logic (Schreiner, 2019). In contrast to interactive proof assistants (such as the Theorema system described below), RISCAL requires no assistance, but is able to fully automatically check the validity of theorems and the correctness of algorithms.

RISCAL is an educational software (Schreiner, 2019) in contrast to other software, such as TLA (Cousineau et al., 2012), of similar design, thus motivating our use of the software. As an educational software, its intended use is to give insight into the meaning and purpose of first order logic a more expressive but also more difficult language than propositional logic (the domain of SAT & SMT); e.g., first order logic is able to describe the complex relationships required in mathematical theories or in the formal specification of computer programs. In particular, RISCAL can quickly demonstrate that an attempted first order logic formalization is (due to errors and omissions) *not* adequate; this is actually the problem that students (and experts) have to deal with most of the time but that is not well addressed by proof-based tools (the construction of proofs is tedious and the inability to derive such a construction does not necessarily demonstrate that the goal formula is invalid).

RISCAL has already successfully supported courses on the formal specification and verification of computer programs at our university (Schreiner, 2019). In our course it was used as a learning aid during the first half of the module 2. In detail, students were issued three bonus assignments consisting of prepared specification templates in which students had (as demonstrated by corresponding examples) to fill in the missing parts of formalizations; after each step they could apply the RISCAL checking mechanisms to determine whether their entries were adequate. The correctness of submissions could thus be completely self-checked before actually handing them in; teaching assistants mainly verified their plausibility.

RISCAL was distributed in the form of a pre-configured virtual machine to be executed on the students' own computers; videos were prepared to describe the installation of the use of this software. While we experienced few technical problems, nevertheless some students may have shied away from the use of the software, because of the technical requirements and/or the mode of interaction with it (which required the manipulation of text files). Alternative solutions are being considered for future iterations of the course, such as an (already developed) web-based exercise interface.

3.3 The Theorema Proof Assistant

Theorema is a mathematical assistant system (Buchberger et al., 2016) based on the well-known Software system Mathematica (Inc.,). Theorema is freely available (Buchberger et al., 2016) (GNU GPL), students need Mathematica installed, which is affordable since many universities own a campus-license of Mathematica. Theorema is a Mathematica package, i.e. it just consists of a folder to be copied to the right location. By design, Theorema aims to support the mathematician during all phases of mathematical activity. While this serves as a philosophical goal, the current implementation of Theorema can *compute expressions* built-up by numbers, tuples, and finite sets and *automatically prove* statements expressed in Theorema's version of higher order predicate logic. Computation is a useful tool for checking adequacy of definitions, because *checking* its behavior on several finite cases can give certainty that the definition as written fits. This is similar in spirit to what RISCAL (Section 3.2) does when checking specifications. However, Theorema was used in our course mainly for its proving capabilities. We consider correct logical argumentation and doing (simple) mathematical proofs an important competence being taught to computer science students. According to our philosophy, a thorough understanding of "mathematical proving" as a (mainly) syntactical process on sets of predicate logic formulas (hypotheses and the proof goal) determined by their syntactical structure helps students doing their own proofs. The goal of using a theorem prover in this setting is *not* to convince students *that* certain statements are true. Rather, they should learn from the prover *how* it proved some theorem. We consider the automated prover as a proof tutor, and students can train as many examples as needed.

The key feature of Theorema is that it generates human-readable proofs using inference rules inspired by natural deduction. It is though not a classical natural deduction calculus as explained in literature and taught in our course because Theorema aims at human-like proofs (in contrast to e.g. a minimal set of inference rules). Theorema was used in the proving section of the "First Order Logic" module. Students were offered three bonus exercises with increasing difficulty, where in each exercise they received a Theorema notebook with a theorem already contained in there. Exercise 1 was quantifier-free, Exercise 2 contained alternating quantifiers, and Exercise 3 needed auxiliary definitions that were also part of the notebook. In all three, it was possible to get an automated proof of the theorem without further configuration of the system. (Note that, in general, a user can fine-tune the prover by switching on/off certain rules and by setting rule priorities.) The bonus exercises were meant as a preparation for the Theorema Lab Exercise, where the task was to *first use Theorema* to prove a theorem *and then prove* the theorem with pencil and paper, being of course allowed to use the Theorema-proof as a model.

4 Evaluation and Results

In this section, we cover the design of our questionnaire, the evaluation, and the results.

4.1 Questionnaire Design

A typical end-of-semester questionnaire (usually referred to as a course evaluation) is used to evaluate the effectiveness of the lecturer, the overall presentation of the course material, and the evaluation method (grading system) of the students. The center for teaching and learning at UC Berkeley provides an outline of a typical

end-of-semester evaluation[1]. Such evaluation forms usually include generic questions such as "The course (or section) provided an appropriate balance between instruction and practice?" which are to be answered by selecting from a discrete scale.

There has been a large number of investigations into various aspects of such evaluation forms, as covered by Nicole Eva (Eva, 2018) and even journals dedicated to the subject (Spooren and Christiaens, 2017) (an interesting paper from such a venue). While design and execution does come into question (McClain et al., 2018) much of the literature is concerned with effectiveness and bias. In our case, the effectiveness of the questionnaire was not of the highest priority because we were not using it for evaluation, rather we want the opinion of the students as to how the overall presentation affected their understanding of the material. As mentioned in (Lee et al., 2018) we wanted to gain an understanding of the student's conceptual gains and their relationship to the presentation. Thus, our foremost goal was to make the questionnaire engaging enough to get the students to complete it. Towards increasing student engagement we decided to make our questionnaire mostly free-form. As documented in literature, this design choice increased the difficulty of analysis. See Figure 1 for the first two questions of eight questions.

Instead, we asked students to draw curves for understanding vs. time and interest vs. time. Additionally, they were provided the opportunity to mark three points on the curve which were significant to their experience. The macrostructure of the course allows the students to have reference points, i.e. three mostly distinct modules, thus allowing easier evaluation.

The rest of the questions included in the questionnaire concern the inclusion of software within the course and how the software was included in the course. For the past four years, each iteration of the course introduced software as either a bonus exercise or as a part of a lab assignment. Questions 5, 6, and 7 were designed to see if the software and its integration into the course is mature enough to make it an explicit part of the grade, i.e. students may also lose points for not completing assignments using the software. The final question concerns students' thoughts about an experimental game-based approach to teach propositional satisfiability.

4.2 Evaluating Student Illustrations

Rather than asking students to provide a written description of there experiences we ask them to draw curves depicting their understanding and interest over the course of the semester. Space was left (see Figure 1) for the students to provide a short description of any particularly important parts of the course which influenced their understanding or interest. Coordinate grid provided to the students was divided into three sections, one for each module of the course. Out of the 134 questionnaires handed in, 131 of them had illustrations in the provided charts, roughly $\sim 98\%$ of the questionnaires[2].

We evaluated the curves drawn by students by first placing them into 5 catagories: **Constant**, **Linear**, **Parabolic**, **Saw Tooth**, and **Other**. Furthermore per module, we considered the **Slope** (either **-1**,**0**, or **1**), the **Maximum** (either **0**,**.25**,**.5**,**.75**, or **1**), the **Minimum** (either **0**,**.25**,**.5**,**.75**, or **1**), **Jump** (either **yes**,**no**), and **Drop** (either **yes**,**no**).

Using these measures we were able to define the **Concavity** of the curves, i.e. slope between modules. Note Students where not provide with a discrete scale on the y-axis thus, we had to apply the scale during evaluation. This scale spanned the interval $[0, 1]$ in .25 increments. To denote significant changes in value, we added

[1] https://teaching.berkeley.edu/course-evaluations-question-bank

[2] For more details see the technical report: https://ww w3.risc.jku.at/publications/download/risc_5885/Report.pdf.

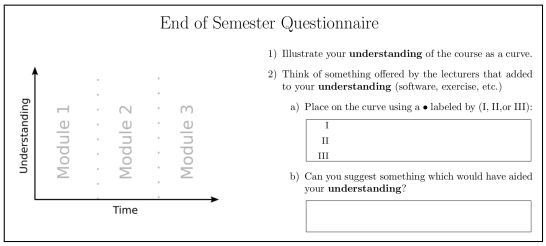

Figure 1: The first two questions of our questionnaire. The left side provides space for student illustrations. The right side provides space for highlighting important activities.

the **Jump** and **Drop** features which state whether a module contains a significant positive change (**Jump**) or negative change (**Drop**)[3].

4.3 Questionnaire Results

Understanding and interest in the course where highly correlated and thus we focus on student illustrations of understanding. The majority of students drew parabolas with a minimum within module 2, i.e. a majority were concave down. The second most common curve type was saw-tooth with a minimum in module 2 with a further drop occurring in module 3. Note module 3 is essentially first-order logic without quantification, i.e. term manipulation rather than explicit proof construction. Overall, the majority of students understood less and were less interested as the course progressed. The significant number of saw-tooth illustrations points to issues with the computational aspect of formal reasoning being difficult for students to grasp. This may be the first time many of the students have seen formal reasoning. Our survey provides preliminary evidence that developing tools and software for these topics would be a worthwhile endeavor. Concerning proof construction, systems have been developed already, using COQ (Knobelsdorf et al., 2017) and Z3 (Ehle et al., 2017). However, to the best of our knowledge, no one has focused on educational tools for the computational aspects of first-order reasoning.

Together, these results account for $\sim 66\%$ of the illustrations for understanding. Many students had praise for the portion of module 2 focusing on the application of first-order reasoning to software verification aided by RISCAL. The particularly problematic topic was proof tree construction which invariably required sound application of formal reasoning, i.e. logic without direct application. It is precisely this approach to formal reasoning which resulted in a loss of student interest and understanding.

Many students mentioned that lab sessions and the introduction of software increased their interest in the course, but according to the illustrations this did not necessarily mean they understood more. The software was mainly introduced as part of the weekly challenges, When asked why they completed a weekly challenge, the most common answer was for credits (one of the problems with extra credits (Norcross et al., 1993; Pynes, 2014)). Some students pointed out their desire to increase their understanding. It is not clear if the software inadvertently had some effect on student understanding. This not been investigated.

[3] A spreadsheet containing all results may be found here:
https://www3.risc.jku.at/publications/download/risc_5885/FINALRESULTS.xlsx

5 Lessons Learned

Clearly *computational logic* is an essential part of *computational thinking*. There is an increased need to equip students of computer science and related fields with the important skill of applying logic to formalize system properties and reason about software and hardware systems. The introduction of our Logic course as a mandatory course in the Bachelor curriculum is the reaction to a request by the applied faculty in our department. The goal was to educate students in such computational aspects of logic. Our own experience in applying formal technology in Industry confirms this view. A fundamental understanding of the algorithms of logical reasoning is also important to efficiently apply logical reasoning tools in practice. It turned out that the main challenge is how to teach these skills, focusing on the practical aspects of logic, ignoring classical more abstract concepts, less important from the computational point of view.

We addressed this challenge by (1) emphasizing concepts most relevant in practical applications of logic, (2) letting the students gain hands-on experience in using automatic and computer-assisted logical reasoning tools, and (3) providing immediate feed-back through weekly mini-tests, weekly challenges, and lab exercises. To assess the effect of our measures we presented students with a questionnaire at the end of the semester. The results show that the last two measures are very effective. Students appreciate the way we teach logic on concrete problems too.

As expected the more classical part of the lecture was considered the most difficult one. We are trying to make it more accessible in the future. In particular, we already decided to completely remove the discussion of complexity and decidability and rely on later courses in the curriculum to cover these (important) concepts. We are further investigating how to use games or puzzles to introduce quantifiers. Another big concern of the students was the usability of the software. To address this issue we will expand the use of web technology and will work on an app-based approach too. But in general, we got the impression that the course was very well received by the students. It produced low-drop rates compared to other courses without compromising on quality.

For us as teachers, the most striking lessons are as follows. First, even though evident in hindsight, you can teach advanced modern logical reasoning techniques such as SAT and SMT in the first semester. Students seem to like the hands-on experience they get with these topics. Second, automatically checking first-order logic properties on simple programs makes first-order logic much more accessible. Third in an introductory course on Logic, one should remove all the more abstract and philosophical topics of logic and post-pone it to later more specialized courses. Fourth, and most unexpected, we learned about the importance of presentation to usability of the introduced tools. What seems simple to us in terms of usability may be a barrier for students, for example, the difficulty of installation, the complexity of the interface, etc.

Finally, we concluded that computational logic is much simpler to understand and teach than programming. In our experience, this applies not only to first-year students of computer science but also to other fields and on all levels of education. As a consequence, we suggest that attempts to spread computational thinking throughout society should not focus on teaching programming skills only. Logic is essential too and at least with the focus on computational logic we propose it is easy to integrate successfully into a technical curriculum.

Acknowledgements

Suported by the LIT LOGTECHEDU project and the LIT AI Lab both funded by the state of upper Austria.

References

Barland, I., Felleisen, M., Fisler, K., Kolaitis, P., and Vardi, M. Y. (2000). Integrating logic into the computer science curriculum. In *Annual Joint Conference on Integrating Technology into Computer Science Education*.

Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009). Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, Amsterdam, The Netherlands.

Ben-Ari, M. (2012). *Mathematical Logic for Computer Science*. Springer, London, 3rd edition.

Böhne, S. and Kreitz, C. (2017). Learning how to prove: From the coq proof assistant to textbook style. In *Proceedings of ThEdu@CADE'17*, pages 1–18.

Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., and Windsteiger, W. (2016). Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *JFR*, 9(1):149–185.

Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. (2015). Moving fast with software verification. In *Proceedings of NFM'15*, volume 9058 of *LNCS*, pages 3–11. Springer.

Chen, Y.-A., Clarke, E. M., Ho, P.-H., Hoskote, Y. V., Kam, T., Khaira, M., O'Leary, J. W., and Zhao, X. (1996). Verification of all circuits in a floating-point unit using word-level model checking. In *Proceedings of the 1st Int. Conference on Formal Methods in Computer-Aided Design, FMCAD'96*, volume 1166 of *LNCS*, pages 19–33, Cham, Switzerland. Springer.

Cook, B. (2018). Formal reasoning about the security of amazon web services. In *Proceedings of CAV'18*, volume 10981 of *LNCS*, pages 38–47, Cham, Switzerland. Springer.

Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., and Vanzetto, H. (2012). TLA+ proofs. In *Proceedings of FM'12, Paris*, volume 7436 of *LNCS*, pages 147–154, Berlin, Germany. Springer.

development team, T. C. (2019). The coq proof assistant reference manual. Version 8.10.

Edgington, J. M. (2010). *Toward Using Games to Teach Fundamental Computer Science Concepts*. Doctoral dissertation, University of Denver, USA.

Ehle, A., Hundeshagen, N., and Lange, M. (2017). The sequent calculus trainer with automated reasoning - helping students to find proofs. In *Proceedings of ThEdu@CADE'17*, pages 19–37.

Eva, N. (2018). Annotated literature review: student evaluations of teaching (set). Technical report, University of Lethbridge Faculty Association, Canada.

Eysink, T. (2001). *Signs for logic teaching*. PhD thesis, University of Twente, The Netherlands.

Gallier, J. H. (1985). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, Inc., New York, NY, USA.

Goldson, D., Reeves, S., and Bornat, R. (1993). A review of several programs for the teaching of logic. *Computer Journal*, 36(4):373–386.

Gries, D., editor (1981). *The Science of Programming*. Springer, New York, USA.

Hooper, A. (2017). A serious game for teaching first order logic to secondary school students. Tr, Department of Computer Science, University of Bath.

Huertas, A. (2011). Ten years of computer-based tutors for teaching logic 2000-2010: Lessons learned. In *Proceedings of the 3rd Int. Congress Conference on Tools for Teaching Logic, TICTTL'11*, pages 131–140, Berlin, Heidelberg. Springer.

Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, Cambridge, UK.

Inc., W. R. Mathematica, Version 11. Champaign, IL, 2018.

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA, USA.

Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V. A., Reeber, E., and Naik, A. (2009). Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In *Proceedings of CAV '09*, volume 5643 of *LNCS*, pages 414–429. Springer.

Kaufmann, M., Manolios, P., and Moore, J. (2000a). *Computer-Aided Reasoning: ACL2 Case Studies*, volume 4 of *Advances in Formal Methods*. Springer, New York, NY, USA.

Kaufmann, M., Manolios, P., and Moore, J. (2000b). *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Springer, New York, NY, USA.

Knobelsdorf, M., Frede, C., Böhne, S., and Kreitz, C. (2017). Theorem provers as a learning tool in theory of computation. In *Proceedings of the ICER 2017*, pages 83–92, Tacoma, WA, USA.

Leach-Krouse, G. (2017). Carnap: An open framework for formal reasoning in the browser. In *Proceedings of ThEdu@CADE 2017, Gothenburg, Sweden, 6 Aug 2017.*, pages 70–88.

Lee, E., Shan, V., Beth, B., and Lin, C. (2014). A structured approach to teaching recursion using cargo-bot. In *Tenth Annual Conference on Int. Computing Education Research, ICER '14, Glasgow, Scotland, UK, April 11–13*, pages 59–66. ACM.

Lee, L. J., Connolly, M. E., Dancy, M. H., Henderson, C. R., and Christensen, W. M. (2018). A comparison of student evaluations of instruction vs. students' conceptual learning gains. *American Journal of Physics*, 86(7):531–535.

Lethbridge, T. C. (2000). What knowledge is important to a software professional? *Computer*, 33(5):44–50.

Makowsky, J. (2015). Teaching logic for computer science: Are we teaching the wrong narrative? In *Proceedings of the 4th Int. Conference on Tools for Teaching Logic, TTL 2015*, Leibniz Int. Proceedings in Informatics, pages 101–110, Dagstuhl, Germany. Dagstuhl Publishing.

Makowsky, J. A. and Zamansky, A. (2017). Keeping logic in the trivium of computer science: A teaching perspective. *Formal Methods in Systems Design*, 51(2):419–430.

McClain, L., Gulbis, A., and Hays, D. (2018). Honesty on student evaluations of teaching: effectiveness, purpose, and timing matter! *Assessment & Evaluation in Higher Education*, 43(3):369–385.

Muratet, M., Torguet, P., Jessel, J.-P., and Viallet, F. (2009). Towards a serious game to help students learn computer programming. *Int. Journal of Computer Games Technology*, 2009.

Norcross, J. C., Dooley, H. S., and Stevenson, J. F. (1993). Faculty use and justification of extra credit: No middle ground? *Teaching of Psychology*, 20(4):240–242.

Page, R. L. (2003). Software is discrete mathematics. *SIGPLAN Not.*, 38(9):79–86.

Pynes, C. A. (2014). Seven arguments against extra credit. *Teaching Philosophy*, 37(2):191–214.

Reeves, S. and Clarke, M. (1990). *Logic for Computer Science*. Addison-Wesley, Boston, MA, USA.

Reid, A., Chen, R., Deligiannis, A., Gilday, D., Hoyes, D., Keen, W., Pathirane, A., Shepherd, O., Vrabel, P., and Zaidi, A. (2016). End-to-end verification of processors with isa-formal. In *Proceedings of the Int. Conference on Computer Aided Verification, CAV'16*, pages 42–58, Cham, Switzerland. Springer.

Reinfelds, J. (1995). Logic in first courses for computing science majors. In *Proceedings of WCCE'95*, pages 467–477, Boston, MA. Springer.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence - A Modern Approach, Third Int. Edition*. Pearson Education, Upper Saddle River, NJ, USA.

Schreiner, W. (2019). Theorem and Algorithm Checking for Courses on Logic and Formal Methods. In Quaresma, P. and Neuper, W., editors, *Post-Proceedings ThEdu'18*, volume 290 of *EPTCS*, pages 56–75.

Spooren, P. and Christiaens, W. (2017). I liked your course because i believe in (the power of) student evaluations of teaching (set). students' perceptions of a teaching evaluation process and their relationships with set scores. *Studies in Educational Evaluation*, 54:43–49.

Vardi, M. Y. (1998). Sigcse'98 panel on logic in the cs curriculum. https://www.cs.rice.edu/ vardi/sigcse/.

Wing, J. M. (2000). Invited talk: Weaving formal methods into the undergraduate computer science curriculum. In *Algebraic Methodology and Software Technology*, pages 2–7, Berlin, Germany. Springer.

Zamansky, A. and Farchi, E. (2015). Teaching logic to information systems students: Challenges and opportunities. In *Proceedings of the 4th Int. Conference on Tools for Teaching Logic, TTL'15, Rennes, France, June 9-12*, Leibniz Int. Proceedings in Informatics, pages 273–280, Dagstuhl, Germany. Dagstuhl Publishing.

Zamansky, A. and Zohar, Y. (2016). 'mathematical' does not mean 'boring': Integrating software assignments to enhance learning of logico-mathematical concepts. In *Advanced Information Systems Engineering Workshops*, pages 103–108, Cham, Switzerland. Springer.