Logic in Computer Science: tool-based modeling and reasoning about systems

Michael Huth
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506-2302

Abstract - *Recent years have brought about the development of powerful tools for verifying specifications of hardware and software systems. By now, major companies, such as Intel, IBM, AT&T, Siemens, and BT have realized the impact and importance of such tools in their own design and implementation processes as a means of coping with the ever-increasing complexity of chip and software designs. This necessitates the availability of a basic formal training that allows* Undergraduate *students to gain sufficient proficiency in using and reasoning with such tool-animated frameworks. We present an existing course, "Logical Foundations of Programming", that aims at meeting these educational goals. After describing inherent challenges that such a course faces, we then evaluate this course in the larger context of what logical frameworks, if any, should be taught and where they may be placed in a computer-science related undergraduate curriculum.*

I. Our course

A. Mission statement and course description

The course *CIS301 Logical Foundations of Programming*, initiated by Dr. David A. Schmidt and required of our Computer Science and Information Systems majors since 1993, has as current objectives [14] to

"study the fundamentals of symbolic logic: how to write proofs and how to reason semantically. We apply those acquired techniques to model computer systems and to check whether these models satisfy formally specified properties. We further use these skills for the design and verification of algorithms."

The topics we chose for acquiring the knowledge and skills above are [14]:

1. *Logic as in "general education".* We study propositional logic as the prototypical logic formalism. We ask: 'What constitutes valid arguments? How can we demonstrate that certain "inferences" have a valid argument?' We design and utilize a proof calculus (natural deduction) and a semantics based on truth tables to address these issues (3-4 weeks).

2. *Predicate logic.* The design and use of database query languages and their supporting tools depend on a thorough understanding of the syntax and semantics of formulas with quantifiers such as "there exists somebody who is older than forty, drives a Lexus, and likes Thai food", or "all taxpayers of Manhattan are Wildcat fans". The realization of polymorphically typed languages and general data abstraction or encapsulation facilities, or software specification languages also require mechanisms for which predicate logic provides a sound design framework. We study the syntax and semantics of predicate logic and understand how to evaluate its formulas over their corresponding notion of models. We also discuss its proof theory and prove important quantifier equivalences. (3-4 weeks).

3. *Verification by model checking.* Protocols, networks, and distributed systems can generally not be described by code of some deterministic programming language. Such systems exhibit concurrent behavior and they are typically *reactive* in the sense that their behavior depends on what the environment can offer (e.g. "Is the printer busy?"). Computation tree logic (CTL) is currently one of the popular frameworks used in verifying properties of concurrent systems. We study its syntax and semantics, and use those insights to design an automated verification algorithm which takes a description of a system and specifications of expected behavior as input and checks whether that system meets those expectations. That algorithm is the foundation for a tool, the symbolic model verifier (SMV) [20], which we use to evaluate some basic designs, e.g. simple elevator systems and a mutual exclusion protocol (3-4 weeks).

4. *Program Verification.* Given a program P, written in some imperative programming language with if- and while-statements, what can we say about its input/output behavior without having any run-time information about its variables? We develop a proof calculus for verifying triples $(\!|\phi|\!)\ P\ (\!|\psi|\!)$, where ϕ and ψ are statements about the store (e.g. describing the values of variables) before, respectively after, the execution of a program P (2-3 weeks).

5. *Binary Decision Diagrams.* We explain how Boolean functions and their logical manipulations can be implemented, often very efficiently, as binary decision diagrams [3], [4], [5](1-2 weeks).

After successful completion of this course, we expect students to be able to *actively* apply logical frameworks in modeling and reasoning about computer systems, be they realized in hardware, implemented in software, or embedded in technology that interacts with some open, non-controllable, environment.

B. Prerequisites and background

Students are expected to have completed an introductory programming course. About one third of all enrolled students have also taken a second programming course on algorithms and data structures. However, about 15% of the enrolled students do not major in computer science or a related discipline, though they mean to switch to such majors; their background ranges from history, chemical and mechanical engineering, to business, biology, and horticulture majors. This is problematic as the course is customized toward freshmen or sophomores with a computer-science related major.

C. Mission accomplished?

The challenge that most profoundly threatens a successful realization of the objectives stated in Section I-A is that one would need at least twenty weeks to adequately cover all topics above. Consequently, one has to de-emphasize certain aspects and techniques which students hopefully will master sufficiently when meeting them again in subsequent courses. The identification of which points to skim should be guided by the intent of the mission statement, i.e. *omissions should not critically impair students ability to apply logical frameworks to model and reason about systems*. Therefore, our current approach is to skip the *justification* of meta-results, but, whenever possible, to state those results and reflect on their significance. For example, we do not prove the soundness and completeness of a natural deduction calculus for propositional logic, but we discuss its meaning (e.g. soundness as a design validation technique and as a tool for showing the non-existence of a proof). This compromise re-instantiates the old computer-science debate about how much of the "what?", the "why?", or their potential interdependencies one should teach. Related to that, it also raises the familiar issue of whether one should bundle such topics into a single course, be it early or rather late in the curriculum, or whether one should teach these topics exclusively in those courses that actually apply them. We return to these important questions in Section IV.

We often teach this course by *omitting*, among other things, a proof of the undecidability of validity in predicate logic, a discussion of intuitionistic propositional logic, the design and correctness proof for a labeling algorithm that is the basis for the verification tool SMV, a discussion of a more complex concurrency protocol (the alternating-bit protocol), more advanced temporal logics (e.g. CTL*), program termination arguments that rest on more complex well-founded orderings, relative completeness issues for Floyd-Hoare style programming logics, and the important application of binary decision diagrams for building the model checking tool SMV. While some of the omitted topics may be too advanced for a sophomore level, their absence from our course basically means that they may not be discussed in the remainder of students' program of studies. In a more tightly integrated curriculum, one would introduce this more advanced material at the appropriate places of subsequent courses.

II. Connections to other courses

The typical contents of our course establish connections, be they implicit or explicit, to other courses that are usually required in a computer-science curriculum; these topics also create opportunities for an improved integration of concepts that cross undesirable dividing lines of individual courses.

A. Algorithms and data structures

We cover pseudo-code for a recursive, and staged, algorithm that computes for any propositional logic formula an equivalent formula in conjunctive normal form. The design and verification of the design features recursion, mathematical induction, grammars in Backus-Naur form, and complexity issues pertaining to the satisfiability and validity of formulas. The algorithm is staged in that it first transforms away all implications, then pushes all negations down to the level of atoms, and finally solves the problem for formulas of the much simpler grammar $\phi ::= L \mid \phi \wedge \phi \mid \phi \vee \phi$, where L ranges over literals (atoms and their negation). One could conceivably cover the design and verification of algorithms for satisfiability, such as Horn formulas, or heuristic SAT solvers, such as Stålmarck's method [23], [22]. In predicate logic, one could feature algorithms for computing certain quantifier normal forms.

Exposure to a labeling algorithm for finite-state verification illustrates depth-first backwards search in a directed graph; this search is recursive and the recursion is driven by the logical structure of the specified behavior, written as a CTL formula. The description and evaluation of small designs with the tool SMV makes students appreciate how such graphs can be modeled with a modular guarded-command language with non-deterministic assignment. The discussion of program logics contains the linear algorithm for computing minimal-sum sections of integer arrays [12] as a case study. Finally, binary decision diagrams [3], [4], [5] require algorithms that implement the familiar logical operations on such diagrams. Some of these algorithms illustrate dynamic programming at an accessible level.

B. Programming languages and their design

The design of a program logic for a simple imperative language challenges students to reason about "method-like" code they would use within objects in the object-oriented programming paradigm. They learn how to compute necessary preconditions to ensure desired postconditions, based on a simple heuristics for finding and validating useful invariants for while-statements. The tool SMV exposes them to a domain-specific programming language for describing finite-state systems.

The natural deduction system for propositional logic, as presented in [16], encourages students to think of conditional judgments and the stating and discharging of temporary assumptions as a statically scoped language with local declarations, whether they know such terminology, or not. Although freshmen and sophomores may only have an intuitive understanding of type checking and type inference in programming languages, they by and large appreciate that these concepts correspond to "grading" their proofs and "finding" such proofs, respectively. These connections to language design methodologies carry over to the extended natural deduction system for predicate logic, dealing with *hypothetical* and *parametric* judgments. [1]

The final point may seem too trivial to be mentioned here, but it is of paramount importance and seems to pose the most common conceptual challenge to students who take this course. We refer to the ability to read and understand a general syntactic or semantic rule scheme and to apply it consistently in a concrete instance by matching the patterns of the rule with those of the given instance. For example, natural deduction has two rules for introducing disjunctions. One of them says that we can prove $\phi \vee \psi$ by first proving ϕ. It is crucial to realize that this makes ϕ bound in $\phi \vee \psi$, but ψ can be chosen to one's liking. Similar binding issues occur with all other rules of that calculus and its extension to predicate logic. We are often surprised at how students find the process of pattern matching difficult and how they frequently struggle with parts of proofs that are uniquely

[1] Our course does not use such technical terminology, but students learn these concepts implicitly as *operational activities*.

determined by the given rules.

C. Compilers and flow analysis

While compiler courses do not take a prominent place in todays undergraduate curriculum, the discussion of a model checking platform and its temporal logic could, at a later stage, be enriched by investigating the connections between conventional flow analyzes, e.g. those used during the optimization phase of a compilation, and model checking. It is well known that model checking tools can be used for bit-vector based dataflow analyzes of programs. On the other hand, one can conceive established techniques for flow analysis as being model checks based on safe abstractions of the underlying program graph; see [21] for a technical reference. Admittedly, such discussions may be confined to graduate programs, but they also point to fruitful connections to traditional topics in automata theory and formal languages.

D. Discrete mathematics and mathematical logic

Discrete mathematics courses for computer-science related majors are most often taught as service courses by mathematics departments. Even if we assume that enrolled students are all computer-science related majors — which may not be the case for the vast majority of institutions, the outlook of such a course and its covered techniques may be ill-suited for preparing our students for a variety of reasons:

1. Such a course may not adequately stress the computational nature and the fundamental algorithms associated with the discrete structures it features. For example, little is achieved if students know what a conjunctive normal form is, but if they do not know how to transform any formula into such a form and why such a transformation may be problematic.
2. The logic component of such a course may present logical formalisms that are remote from the connections to the computer-science issues aforementioned. For example, using a Hilbert-style presentation for reasoning about propositional logic is likely to alienate a typical computer-science student.
3. The course may stress independence results over undecidability results, reflecting the likely bias of a mathematician, versus that one of a computer scientist.
4. Mathematical induction may be presented and applied only to structures that are "indigenous" to mathematics, such as integers and polynomials, without seeing this principal in action on more general recursively generated syntax, such as trees, lists, parse trees of formulas, etc. Students may never see an example of a context-free grammar in such a course.
5. A traditional mathematical style of presentation may introduce a lot of technical terminology that the student will never utilize, and possibly not appreciate, in subsequent courses. For example, students may learn about "the law of non-contradiction" and something called "modus ponens", the first being the *name for a theorem* ($\neg(\phi \wedge \neg\phi)$) which students are able to prove, if needed, the second being the *elimination rule for implication*; each logical operator comes with elimination (data decomposition) and introduction (data construction) rules. In contrast, our course introduces a carefully chosen core of essential terminology; further technicalities are presented only where they are required by the applications.
6. The overall "culture" of such a course may make it difficult for students to take the subject matter seriously and to later transfer their acquired knowledge to courses in their field.

E. Hardware architecture and networking courses

The discussion and use of a model checking framework on simple concurrent systems can be re-activated in a basic networking course, when liveness and safety issues of concurrent protocols arise. Similarly, hardware design and architecture courses can make use of this verification framework and the material on binary decision diagrams to model and analyze simple circuits.

F. Software engineering

An advanced undergraduate software engineering course may focus on methodologies for specifying and analyzing software artifacts in a very general sense. For example, the Object Constraint Language (OCL), a subset of the industry standard Unified Modeling Language (UML), enables developers to write constraints over object models, allowing for the formulation of specific and complex rules that govern todays business models. Apart from the study of design patterns, software architectures, the SCR specification method, etc, and the use of theorem-proving tools, one can present the model-checking methodology as an additional, complementary or synergistic, approach. For example, Marsha Chechik at Toronto University already features this methodology in such a course [6]. Tools for model checking software are beginning to appear. For example, Bandera [1], [9] is a tool-set for translating Java programs to the input of existing model checkers, such as SMV. As these tools mature, they may be introduced into undergraduate courses as a powerful means of testing and verifying (concurrent) programs.

III. Why tools are needed

A. The hype

The applied component of our course introduces, at an accessible level, the model-checking framework SMV that finally has made its way from research papers to a suitable textbook [16]; see also [7] for a more comprehensive and ambitious graduate textbook on the model-checking methodology. Reasoning about the security and reliability of reactive systems represents a major technological challenge that can no longer be adequately met by ordinary scenario-driven testing methodologies. Since such automated verification frameworks are at present a hot topic in industry, graduates fluent in this material are highly sought. SMV, developed by Ed Clarke's PhD student Kenneth McMillan in 1992 at CMU, and related tools are, e.g., used in the development of Intel's, IBM's, and AMD's low- and high-end chips; in NASA's verification of autonomous control systems that use automated reasoning techniques to provide high levels of autonomy and adaptability, part of the Deep Space projects; and are now routinely used in industry at large.

B. Taking the hype aside

Getting a job at NASA, INTEL, or Bell Labs based on the mastery of such tools for the design and evaluation of computer artifacts is something that may seem promising to students from top U.S. and international schools. Undergraduate students who are already determined to pursue a Master or Ph.D. degree in computer science may also be well motivated to actively learn the use of such tools. One has to keep in mind, though, that our main educational constituency is comprised of people who want a "nice, well paid, and perhaps not *too* challenging" job in the *service-related* IT industry. Although Microsoft is currently looking into the application of static-analysis tools on software products *prior* to their beta-release, it is unlikely that such tools will enter the workplace of a typical software engineer in the near future. So why should we teach logic based on such tools at all? To provide a richer context, we first list some representative and well-argued *concerns about the use of tools in such a course*:

1. If the tool is for finite-state verification, it typically has three vital components: a system-description language, a behavior-specification language, and a decision algorithm (hopefully with a debugging facility for negative replies). While students may think of the latter as a black box, they need to appreciate the details of the former two and utilize them to formally model the actual systems and behaviors under consideration. At the very least, this process includes the translation of plain English behavioral requirements into the respective specification formalism. Unfortunately, most tools, while often being share-ware, are rather complicated and have system-description languages that are either too expressive or too specific, and their understanding usually requires the detailed reading of actual research papers, making this generally into a matter for graduate school, at best. The use of SMV in conjunction with [16] and the on-line repository of *specification patterns* [11], however, may mitigate or even avoid such problems altogether.
2. If the tool is a theorem prover or a proof editor, students are likely to struggle with subtle or not so subtle differences in syntax and presentational means between a tool and the conventional nomenclature of underlying concepts as discussed in a textbook. In a more radical approach, one could conceivably teach, say, propositional logic by *only* exposing students to an appropriate tool, but the supplementary use of a white board in class and the usage of tools that are customized for other tasks (e.g. model checking) re-introduce the need for transfer skills that identify concepts across different representational and notational media. This is certainly problematic at the sophomore level. Our current course does not make use of any supporting tools for natural deduction for the reasons cited.
3. A tool may give students a false sense of mastery of a subject matter. For example, theorem provers for propositional logic are likely to be able to prove most classroom problems without requiring user interaction. Even if some interaction is required, little is achieved if students know how to "run" the tool, but if they are not able to produce the "deterministic" parts of proofs on their own.
4. Tools may gain an authority, through a history of usage, that lacks any foundation. Saying that a protocol correctly implements a mutual-exclusion lock *"because the tool said so"* is useless at best, if not outright dangerous. Tools only allow the evaluation of mathematical *models* of systems, so students need a fundamental understanding of a tool's mechanisms if they want to re-translate a tool's findings on the actual system, e.g. by re-interpreting debugging information pertaining to a discovered design flaw, in a meaningful way. This concern also applies to *positively verified* models, as the actual system may be flawed despite the flawless model; the flaw may well reside in the *modeling process*, which tools can only guide and support to some degree.
5. The use of powerful tools may make it difficult to assign homework problems that are neither too advanced nor "trivial", giving the capabilities of the tool(s) in question. For example, using tools for finite-state automata, one may find it hard to test students abilities of reasoning, from first principles, whether two regular expressions are equivalent. Two obvious alternatives are to "trust" students' claim that they don't use the tool for such assignments, or the assignment of more complex and sophisticated projects that cannot be done without the tools, but that also need human interaction and guidance for their successful conclusion. None of these options are compelling in an undergraduate setting.

Second, we mention a few salient *reasons for favoring the use of tools in such a course*.

1. Some of the concerns raised above may not apply if the exposure to such tools happens rather late in the curriculum or if students already had previous exposure and experience with the involved concepts. In that case, one would expect students to move more easily between different representational dialects of the same concepts.
2. Tools, if designed in a thoughtful manner, are generally believed to aid greatly in the visualization and animation of concepts, even more so if these concepts are mathematically abstract entities. For example, letting students "play" with a SAT solver for formulas in conjunctive normal form makes them appreciate what those formulas are and why deciding their satisfiability is such a hard problem.
3. More importantly, as todays hardware and software artifacts are arguably the most complex things ever made by humans, we critically depend on tools to perform tasks that we basically know how to do, but whose scale defies our abilities to correctly carry out these tasks. We cannot check a system with millions or billions of computational states with the exclusive aid of paper and pencil.
4. Tools create an opportunity to establish well defined interfaces of usage between different courses. A tool that was used in a basic logic course, say, can be re-activated in a subsequent networking or software engineering course. Instructors may not have to spend too much time on lecturing about the tool, but can instead focus on the problems meant to be analyzed with the support of such a tool. Throughout a degree program or professional career, knowing a tool could roughly be valued as much as knowing a domain-specific programming language.
5. Most logic-based tools are currently share-ware or may come on a CD with a reasonably-priced textbook; thus, their use does not impose more financial burden on the already strained budgets of computer science departments and their students.

Our course is supplemented by a *worldwide-web tutor* [13] and ancillary online material for instructors [15]. If we consider such on-line resources as additional tools, the pros and cons above largely apply to them as well. In assessing the pros and cons of using logic-based tools in the undergraduate computer science curriculum, it is apparent that a simply yes/no answer is uncalled for. Whether to use a specific tool, or not, depends on multiple factors. For example, are the concepts involved in the tool taught for the first time? Is the tool opaque or can, respectively should, one sketch how it animates the computational concepts in a way that is accessible to undergraduate students? Is a "fully automatic" tool desirable? Does a tool *actively* involve students in the process of modeling and evaluating systems? Etc. In summary, we believe that a judicious choice and placement of logic-based tools can make a computer-science related undergraduate curriculum more integrated and it can demonstrate to students that logical frameworks, their development, and use are one of the main conceptual backbones of their discipline.

C. Other tool options

Jon Barwise and John Etchemendy developed software, with an accompanying manual, that explains predicate logic at work [2]. Using this tool to support existing textbooks potentially causes the compatibility problems we cited earlier. Various freeware or commercial SAT solvers are available over the internet. A competitor to SMV, based on automata-theoretic techniques is SPIN with its system-description language Promela [19]. A verification tool based on a process-algebra formalism is FDR2 [24]. Daniel Jackson and John Chapin use SAT solvers as the computational core engine to debug software architectures that are described as a mix of program interfaces and formal specifications. The language Alloy and its supporting tool, Alcoa [17], are used in the validation of the design of CTAS, a NASA-developed software system for air-traffic control [18]. Among other tools, we use Alcoa and Bandera in our graduate courses on specification and verification within our Master of Software Engineering program [10]. We leave it at mentioning these tools as a small sample of the available support for teaching and using logic-based frameworks; for a more comprehensive list of existing courses and the tools they use see [8]. The choice and placement of such tools, among other things, should be cognizant of the pros and cons discussed in Section III-B.

IV. Should logic topics be bundled?

During a panel discussion in the Special Year on Logic and Algorithms at the Science and Technology Center for Discrete Mathematics and Theoretical Computer Science [25] (DIMACS), Kim Bruce, Phokion Kolaitis, Daniel Leivant, and Moshe Vardi discussed "Logic in the Computer Science Curriculum". Although the panel did not focus on the use of tool-based frameworks, it hinted at many points that have already been addressed implicitly in this paper. As for the proper placement of logic in a curriculum, the panelists don't offer a uniform answer, confirming that this question ought to have answers that are determined by local constraints and objectives.

In fact, placing logic in a computer-science related degree program seems to create an inherent dilemma. In an ideal world, one would introduce the respective logical framework right in that segment, where it is used for the first time. For example, one would teach predicate logic right before or "through" an introduction to a database query language, such as SQL. In that same ideal world, students will also have no difficulties with mastering such formalisms, say, within a week, so that one can swiftly move toward their competent applications. Middle- and lower-tier institutions, are a far cry from such an ideal world. Instead, they often critically rely on having a lower-level course, such as an *Introduction to Discrete Mathematics*, in which one means to expose students to all the logical frameworks they will need in their program of studies. For propositional logic, for example, this typically involves some "hand-waving" about how to prove things, and then maybe a more rigorous treatment of this logic's semantics via truth tables. As such courses are typically "out-sourced" to a mathematics department, instructors of subsequent courses, in which logical frameworks are being used, have little if no knowledge about what *really* happened in that introductory course; or students will simply have forgotten that material, recalling merely an unpleasant and ill-motivated mathematical experience. That way, computer-science degree programs miss a unique opportunity to present logic-based frameworks as a means for modeling and reasoning about computer systems at large. But doing an adequate job at the latter may well require placing these frameworks into the higher-level courses, where they are put to effective usage; whence the dilemma. This dilemma is aggravated by the fact that current computer-science programs, more often than not, seek a reduction of the total number of credit hours or are pressured to teach more "real world" courses. Our freshmen/sophomore, three credit-hour course *Logical Foundations of Programming* attempts to deal with this dilemma in as graceful a way as possible.

V. Conclusion

The problems associated with teaching logic in a computer-science curriculum will not go away with the availability of tool support for logical frameworks. But we believe that the advantages and opportunities of such tools outweigh their potential negative effects within a computer-science program. In light of the recent availability of suitable textbooks, and freely available tools, we encourage American and international computer science degree programs and the relevant professional associations to re-evaluate their need and placement of logic-based frameworks for modeling and reasoning about computer systems in their curriculum designs or recommendations.

Acknowledgments

We want to thank Matthew B. Dwyer and Allen Stoughton for valuable comments on drafts of this paper.

References

[1] Bandera. Extracting safe finite-state models from source code. URL: www.cis.ksu.edu/santos/bandera/, Spring 2000.

[2] J. Barwise and J. Etchemendy. *Tarski's World: Version 4.0 for MS Windows/Book and Disk.* CSLI Publications, January 1994.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[4] R. E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[5] R. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] M. Chechik. ECE 540 - Software Engineering II. URL: www.cs.toronto.edu/˜chechik, Spring 2000.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, January 2000.

[8] Indiana University Computer Science Department. Formal Methods Education Resources: Course Pages. URL: www.cs.indiana.edu/formal-methods-education/Courses/, Spring 2000.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state Models from Java

Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*. IEEE Computer Society Press, June 2000. To appear.

[10] M. Dwyer. CIS 771 - Software Specification. URL: www.cis.ksu.edu/ dwyer/courses/771, Spring 2000.

[11] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998. URL: www.cis.ksu.edu/santos/spec-patterns/.

[12] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1982.

[13] M. Huth. On-line tutor for CIS301 Logical Foundations of Programming. URL: www.cis.ksu.edu/˜huth/lics/tutor/index.html, Spring 2000.

[14] M. Huth. Syllabus for CIS301 Logical Foundations of Programming. URL: www.cis.ksu.edu/˜huth/301/syll.html, Spring 2000.

[15] M. Huth. Textbook home page for CIS301 Logical Foundations of Programming. URL: www.cis.ksu.edu/˜huth/lics/, Spring 2000.

[16] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, January 2000. URL: http://www.cis.ksu.edu/˜huth/lics/.

[17] D. Jackson. An Intermediate Design Language and its Analysis. In *ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, number FSE-6, pages 121–130. ACM SIGSOFT, ACM Press, November 3-5 1998. Lake Buena Vista, Florida.

[18] D. Jackson and J. Chapin. Software Design for Air Traffic Control. URL: www.lcs.mit.edu/research/projects/project?name=9915, Spring 2000.

[19] Bell Labs. On-the-fly, LTL model checking with SPIN. URL: http://netlib.bell-labs.com/netlib/spin/whatisspin.html, Spring 2000.

[20] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[21] D. A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In G. Levi, editor, *Proceedings of the Fifth Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*. Springer Verlag, September 1998.

[22] G. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formulas, January 1994. United States Patent. Patent Number 5,276,897. Date of Patent: Jan. 4, 1994.

[23] G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems (SAFECOMP'90)*, pages 31–36. Pergamon Press, 1990.

[24] Formal Systems. Fdr2. URL: www.formal.demon.co.uk/FDR2.html, Spring 2000.

[25] Rutgers University. DIMACS Symposium: Teaching Logic and Reasoning in an Illogical World. URL: http://dimacs.rutgers.edu/Workshops/Logic/cornellprogram.html#papers, Spring 2000.