'Mathematical' does not mean 'Boring': Integrating software assignments to enhance learning of logico-mathematical concepts

Anna Zamansky[1] and Yoni Zohar[2]

[1] University of Haifa, Israel
annazam@is.haifa.ac.il,
[2] Tel Aviv University, Israel
yoni.zohar@cs.tau.ac.il

Abstract. Insufficient mathematical skills of practitioners are hypothesized as one of the main hindering factors for the adoption of formal methods in industry. This problem is directly related to negative attitudes of future computing professionals to core mathematical disciplines, which are perceived as difficult, boring and not relevant to their future daily practices. This paper is a contribution to the ongoing debate on how to make courses in Logic and Formal Methods both relevant and engaging for future software practitioners. We propose to increase engagement and enhance learning by integrating 'hands-on' software engineering assignments based on cross-fertilization between software engineering and logic. As an example, we report on a pilot assignment given at a Logic and Formal Methods course for Information Systems students at the University of Haifa. We describe the design of the assignment, students' feedback and discuss some lessons learnt from the pilot.

Key words: education, teaching, automated reasoning, software engineering, logic, testing, engagement

1 Introduction

Core mathematical disciplines, such as discrete mathematics and logic, provide the foundations for application of formal methods (FM) in the software engineering domain. Deficient mathematical skills, and as a result, inability to cope with formal notations, are hypothesized as hindering factors for wider adoption of formal methods in industry ([4, 12]). However, the question of how to teach core disciplines to future software engineering and information systems practitioners remains a subject of a fierce debate.

Recent voices [20, 10, 21] call for rethinking the traditional syllabi of mathematical courses, adapting them to the needs of practitioners and making them more tuned towards application of formal methods in the software domain. A notable study in this context is the Beseme project ([14]), which provides empirical evidence that studying discrete mathematics through examples focused on reasoning about software can improve students' programming skills. As pointed

out by [17], software-related examples are also useful for increasing the motivation of students, who can see the applications of the studied material in the domain of their interest. And yet, although the importance of building bridges from logico-mathematical courses to software engineering seems to be widely acknowledged, discussions on practical ways of how this should be done are scarce in the literature.

Empirical studies have shown that students experience more difficulty with concepts from logic than with other computer science topics [1]. Together with a general tendency towards "softening" the teaching of engineering principles noted by [12], it leads to a vicious circle of students' lack of motivation, and perception of logico-mathematical courses as 'boring', 'difficult', 'detached and esoteric'.

In this paper we address the problem of increasing engagement and enhancing learning of logico-mathematical concepts by future software engineering practitioners in a *practical* way. More concretely, we propose to integrate in logic courses 'hands-on' software-related assignments. Such assignments could involve solving a given algorithmic problem with the help of a FM tool (e.g., SAT-solver or theorem prover). Alternatively, it could involve applying SE techniques already familiar to the students to FM systems and tools (e.g., designing a test plan or writing an SRS for a theorem prover). The rationale behind this idea is quite self-explanatory: logical concepts such as satisfiability, theorem, proof system, usually taught in a theoretical, 'paper and pencil'-like way, are transformed and "come to life" in the world of software, made into runnable, testable objects, which feel more real to most SE students. Specifying, validating, running and testing these objects can provide new insights into the nature of these concepts, as the students have a different angle of thinking about them. Having qualitative means to measure success in such assignments may also introduce an element of competitiveness and naturally increases engagement.

As simple as the above idea may sound, the devil is in the details, and the design and realization of such assignments calls for careful consideration with respect to both the educator and the student. Not all logic educators (especially those from math departments) are familiar with the intricacies of software tools. Moreover, many logic-related tools have heavy implementations, involve complex heuristics and may easily cause the "not seeing the forest for the trees" effect. The assignments, therefore, should be easy to replicate and not overflowing with technical details, while still being engaging and fun.

As a first step to implementing the above ideas, in what follows we report on an assignment we have designed and experimented with in two different Logic courses at the University of Haifa. The assignment is based on testing the generic theorem prover Gen2sat.[1] We describe the outcome and the feedback received from students and discuss lessons learnt from running this pilot.

[1] The tool, developed by the second author, is available at `http://www.cs.tau.ac.il/research/yoni.zohar/gen2sat.html`. The implementation is based on the algorithm in [9].

2 Previous Works

Mathematics anxiety is a well-studied phenomenon in education, described as involving feelings of tension and anxiety that interfere with the manipulation of numbers and the solving of mathematical problems in a wide variety of ordinary life and academic situations (see, e.g.,[15, 16]); This phenomenon may be particularly severe in the domain of mathematical logic: empirical studies show that students struggled more with questions related to logic than with those related to other computer science topics (cf. [1]).

In the context of software engineering education, while numerous works discuss teaching of more advanced formal methods, basic logico-mathematical courses have received less attention in this context. Recently, however, more voices are calling for reconsideration of the traditional syllabi in these courses and its adaptation to the needs of future practitioners [19, 11, 10, 21] . As noted by [11], "The current syllabus is often justified more by the traditional narrative than by the practitioners needs." [19] further notes: "...we still face the educational challenge of teaching mathematical foundations like logic and discrete mathematics to practicing or aspiring software engineers. We need to go beyond giving the traditional courses and think about who the target students are."

The Beseme project ([14]) provides an empirical validation to the common belief that studying logico-mathematical courses may improve software development skills. In a three-year study empirical data on the achievements of two student populations was collected: those who studied discrete mathematics (including logic) through examples focused on reasoning about software, and those who studied the same subject illustrated with more traditional examples. An analysis of the data revealed significant differences in the programming effectiveness of these two populations in favor of the former.

Tavolato and Friedrich[17] offers insights into integrating basic formal methods courses at universities of applied sciences, where there are usually limiting factors which are relevant to the IS context as well: (i) students have very limited theoretical background, and (ii) they are strongly focused on the direct applicability of what they are taught. In this context the authors stress the importance of making the practical applicability of the theory understandable to students, and making use of real software-related examples.

3 The Gen2sat Assignment

The first author has been teaching the Logic and Formal Methods course for Information Systems students at the University of Haifa for several years. The course covers introduction to logic and formal specification for the target audience of graduate students, many of whom already work in the industry, and a long time has passed since they took the basic mathematical courses (see [21] for further details on the challenges of logic course design for this target audience).

Looking for means to boost their motivation, we came up with the idea of an assignment that would have the "look and feel" of a software engineering

assignment, so that its domain would be logic. We hoped that "tricking" the students into exercising their SE skills in the subject matter of logic would in fact encourage them to think of logical concepts in a way that would be both fun and beneficiary. Our main challenge was finding the right balance between requiring the students to dive into software technical details to keep the assignment interesting and related to SE, while still emphasizing enough the logical content as their main take-away message. In what follows we describe a concrete way we attempted to address this challenge. We report on the design of our 'hands-on' assignment and students' feedback.

3.1 Gen2sat

The choice of an appropriate software tool for 'hands-on' assignments in logic courses depends on what it is that we want to teach, and how we want to teach it. An interesting direction to consider in this context is the 'logic engineering' paradigm, ([13, 2]) which aims to provide tools for automatic support of investigation of logics. In the spirit of this paradigm, various tools (e.g., MultLog ([3]), TINC ([5]), MetTeL ([18]) etc.) address large families of logics, thereby providing a "bird-view" of such logical concepts as semantics, proof system, axiom, theorem, etc. The tool Gen2sat is also a contribution to this paradigm, aiming to support the use of *sequent calculi* for the specification of logics. Sequent calculi are a prominent proof-theoretic framework, suitable for a wide variety of different logics, and quite a mainstream topic in logic and automated reasoning courses. Most efficient theorem provers based on sequent calculi utilize complex proof search algorithms which require a great deal of ingenuity and heuristic considerations (see, e.g., [6]). In contrast, Gen2sat uses a *uniform* method for deciding derivability using the polynomial reduction of [9] to SAT. Shifting the intricacies of implementation and heuristic considerations to the realm of off-the-shelf SAT solvers, the tool is lightweight and focuses solely on the logical content. For these reasons we chose Gen2sat as a tool for enhancing learning of the concept of sequent calculi. In addition, our deep familiarity with the code allowed us to introduce changes (e.g. planting bugs, changing the way the tool is called etc.) quickly and efficiently.

3.2 The Assignment

After teaching sequent calculi (in a two hour lecture), we introduced Gen2sat in class and explained its functionality and features. The students were then requested to play the role of *testers* of the tool. More concretely, they were requested to provide a test plan (as small as possible) which would cover all possible scenarios the tool could encounter. For a quantifiable measure for success we used a standard approach of measuring code coverage, instructing them to install the Eclemma plug-in for Eclipse ([7]) in order to determine the percentage of code activated for a given input. Thus, basically the students' assignment was producing a minimal test plan that would achieve maximal code coverage.

While writing and analyzing different inputs to the tool, the students would potentially gain insights into the wide variety of non-classical logics defined in terms of sequent calculi.

In the second pilot we went one step further, introducing several easily detectable[2] bugs into Gen2sat code in the style of mutation testing ([8]), and encouraged the students to report as many bugs as they could find.

3.3 Results and Feedback

Eight students participated in the first pilot, all of them ended up submitting[3] test plans which achieved between 70% - 85% coverage, and included non-trivial sequent calculi for different languages. Five students participated in the second pilot, all of them got at least 70% coverage, and two of them revealed two (out of three) bugs.

After submission they filled in an anonymous feedback questionnaire. Several students pointed out that the assignment was helpful in understanding logical concepts, e.g., *"it helped me see the variety of different connectives and rules"*, *"for me thinking of the extreme cases was really illuminating"*. The majority of students found the assignment engaging and fun: *"Really fun and challenging!"*, *"It's like a logical riddle, a game I enjoyed playing."*, *"I was sucked into this assignment and did not quit until I found a bug"*, etc.
It is important to note that after receiving detailed instructions how to install Eclemma and use the code of the tool, no technical difficulties were reported by the students.

4 Summary and Future Research

In this paper we have described a practical way of enhancing the learning of concepts in formal logic using software-related assignments. Although drawing concrete conclusions is still premature, we believe that the pilot reported above is an indication of the potential of integrating hands-on assignments based on an interplay between software engineering and logic. It is our hope that this paper will initiate discourse on collecting and evaluating easy-to-use and shareable teaching resources which could be used in a 'plug-and-play' manner for teaching logico-mathematical concepts in ways relevant for modern software practitioners.

References

1. Vicki L Almstrum. Investigating student difficulties with mathematical logic. *Teaching and Learning Formal Methods*, pages 131–160, 1996.

[2] The were two types of bugs: (i) those which caused unexpected messages to be printed, and (ii) those which produced unexpected results, e.g., refuting an axiom.

[3] Interestingly, seven students employed new connectives with arity greater than 2 and three employed also 0-ary connectives (which indeed increased coverage), although they have not seen any such example in class.

2. Carlos Eduardo Areces. *Logic Engineering: The Case of Description and Hybrid Logics*. Institute for Logic, Language and Computation, 2000.
3. Matthias Baaz, Christian G Fermüller, Gernot Salzer, and Richard Zach. Multlog 1.0: Towards an expert system for many-valued logics. In *Automated DeductionCade-13*, pages 226–230. Springer, 1996.
4. Dines Bjørner and Klaus Havelund. 40 years of formal methods. In *FM 2014: Formal Methods*, pages 42–61. Springer, 2014.
5. Agata Ciabattoni and Lara Spendier. Tools for the investigation of substructural and paraconsistent logics. In *Logics in Artificial Intelligence*, pages 18–32. Springer, 2014.
6. Anatoli Degtyarev and Andrei Voronkov. The inverse method. *Handbook of Automated Reasoning*, 1:179–272, 2001.
7. M Hoffmann and Gilles Iachelini. Code coverage analysis for eclipse. *Eclipse Summit Europe*, 2007.
8. William E Howden. Weak mutation testing and completeness of test sets. *Software Engineering, IEEE Transactions on*, (4):371–379, 1982.
9. Ori Lahav and Yoni Zohar. Sat-based decision procedure for analytic pure sequent calculi. In Stephane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 76–90. Springer International Publishing, 2014.
10. J. Makowsky. Teaching logic for computer science: Are we teaching the wrong narrative? In *Fourth International Conference on Tools for Teaching Logic, TTL*, 2015.
11. Johann A Makowsky. From Hilberts program to a logic tool box. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):225–250, 2008.
12. Dino Mandrioli. On the heroism of really pursuing formal methods. In *Formal Methods in Software Engineering (FormaliSE), 2015 IEEE/ACM 3rd FME Workshop on*, pages 1–5. IEEE, 2015.
13. Hans Jürgen Ohlbach. Computer support for the development and investigation of logics. *Logic Journal of IGPL*, 4(1):109–127, 1996.
14. Rex L Page. Software is discrete mathematics. In *ACM SIGPLAN Notices*, volume 38, pages 79–86. ACM, 2003.
15. Frank C Richardson and Richard M Suinn. The mathematics anxiety rating scale: psychometric data. *Journal of counseling Psychology*, 19(6):551, 1972.
16. Brian F Sherman and David P Wither. Mathematics anxiety and mathematics achievement. *Mathematics Education Research Journal*, 15(2):138–150, 2003.
17. Paul Tavolato and Friedrich Vogt. Integrating formal methods into computer science curricula at a university of applied sciences. In *TLA+ Workshop at the 18th International Symposium on Formal Methods, Paris, Frankreich.*, 2012.
18. Dmitry Tishkovsky, Renate A Schmidt, and Mohammad Khodadadi. Mettel2: Towards a tableau prover generation platform. In *PAAR@ IJCAR*, pages 149–162, 2012.
19. Jeannette M. Wing. Teaching mathematics to software engineers. In *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Montreal, Canada, July 3-7, 1995, Proceedings*, pages 18–40, 1995.
20. Jeannette M Wing. Invited talk: Weaving formal methods into the undergraduate computer science curriculum. In *Algebraic Methodology and Software Technology*, pages 2–7. Springer, 2000.
21. Anna Zamansky and Eitan Farchi. Teaching Logic to Information Systems Students: Challenges and opportunities. In *Fourth International Conference on Tools for Teaching Logic, TTL*, 2015.