When the Student Becomes the Teacher

Marie Farrell[1(✉)] and Hao Wu[2]

[1] Department of Computer Science, University of Liverpool, Liverpool, UK
marie.farrell@liverpool.ac.uk
[2] Department of Computer Science, Maynooth University, Maynooth, Ireland

Abstract. Making formal methods accessible and appealing to future software engineers is vital to promote their uptake in industry and to increase participation in formal methods research. In this paper, we report on our initial experience of both studying and, subsequently, teaching the same software verification module at Maynooth University, Ireland. By analysing on our own teaching and learning experiences along with the students' grades from the 2018–2019 academic year, we present our four initial observations and two hypotheses that we intend to investigate during the 2019–2020 academic year.

1 Introduction

Encouraging students to take an interest in formal methods has generally been perceived as a difficult task [4,5,8–10,18]. Though there has been some success in convincing software developers to use formal methods, it is still quite challenging [20]. In order to increase the uptake of formal methods in industry, we believe that we must first begin by convincing our students that formal methods are useful and relevant for industrial use [4,10,13,17].

In this short paper, we report on our experiences of both studying and teaching the Software Verification module at Maynooth University. We provide some analysis and discussion which we use as a basis for identifying ways to improve this module and to capture the students' interests.

We summarise our contributions as follows:

1. We report on our experience of studying, during our time as undergraduates, and subsequently, teaching a formal methods module to undergraduate students at Maynooth University. To this end, we analyse and discuss this module in light of the associated exam results from the 2018–2019 academic year.
2. We present our observations and form two hypotheses to be further investigated to improve both the teaching and learning experience for this module.

The remainder of this paper is structured as follows. In Sect. 2, we provide an overview of the formal verification module that we both studied and taught

This work is partially supported through EPSRC Hubs for Robotics and AI in Hazardous Environments: EP/R026092 (FAIR-SPACE).

© Springer Nature Switzerland AG 2021
A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 208–217, 2021.
https://doi.org/10.1007/978-3-030-71374-4_11

at Maynooth University. We describe the assessment process for this module in Sect. 3 where we briefly analyse the exam results from the 2018–2019 academic year. In Sect. 4, we reflect upon our own experience both as students and as teachers of this module. We make four observations about the module's current status by combining our reflection with the analysis of the exam results described in Sect. 3. Based on these observations, we form two hypotheses in Sect. 4.3 to be investigated during the current (2019–2020) academic year. Finally, Sect. 5 concludes and outlines future research directions.

2 Module Overview

The Software Verification module (CS357) at Maynooth University aims to provide students with an understanding of both the theoretical and practical applications of formal software verification techniques[1]. The majority of the students taking this module are third-year (Bachelor's degree) students studying Computer Science. For these students, and those studying for the Computational Thinking Bachelor's degree (usually approx. 10 students), this module is compulsory. This module is optional for those studying General Science where Computer Science is a chosen subject. This module assumes that the students have already taken modules in basic Java programming and discrete mathematics (or equivalent).

This module runs over 12 weeks (2 lecture hours and 2 laboratory hours per week) and covers a wide range of different topics. The topics that are covered and the duration spent on each is outlined below:

1. Design by Contract (1 week) [15]
2. Natural Deduction Proofs and the Coq theorem prover (3 weeks) [3]
3. Hoare Logic (2 weeks) [12]
4. Spec# (2 weeks) [1]
5. SAT/SMT (2 weeks) [7]
6. Model Checking (2 weeks) [12]

Upon successful completion of this module, the students should be able to:

1. Explain the role of verification in software engineering.
2. Create mathematically precise specifications.
3. Prove the correctness of programs using Hoare logic.
4. Use different tools to analyse and verify properties of specifications.

These four learning objectives are reflected in the exam structure and continuous assessment (CA) that we describe in the next section.

[1] Full module description is available at: http://apps.maynoothuniversity.ie/courses/?TARGET=MODULE&MODE=VIEW&MODULE_CODE=CS357&YEAR=2020.

Table 1. There are four questions on the exam and each reflects different aspects of the module as outlined in Sect. 2. These questions are designed to assess the learning objectives described in Sect. 2.

Question	Examined topics	Weight (marks)
Q1	Design by contract Propositional and predicate logic Natural deduction proofs	25
Q2	Satisfiability CNF Translation DPLL (Pure literal, Unit clause and Unit propagation)	25
Q3	Hoare Logic	25
Q4	Basic SMT encoding Spec# Programming (Pre/Post conditions, Loop invariants) Linear temporal logic encoding	25

3 Assessment

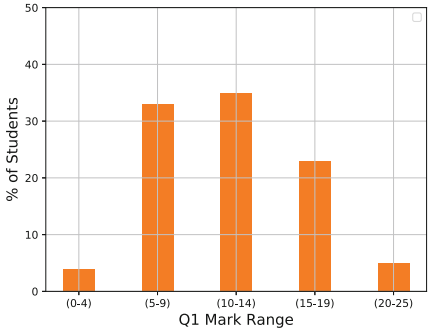
In this section, we describe how this module is assessed. In particular, each student's final grade consists of 30% for continuous assessment (CA) with 70% for the final examination. To obtain CA, each student is required to attend one 2-hour laboratory session every week in order to complete their weekly assignment and to get it graded by one of the tutors. These assignments, 11 in total, are based on the material covered during the lectures each week. The first 3 assignments focus on assessing basic understanding of natural deduction proofs using the Coq theorem prover. The next 2 assignments are based on Hoare Logic. The remaining ones examine a range of verification tools such as Spec# and Z3. At the end of term, the students must complete their final exam.

3.1 Exam Structure

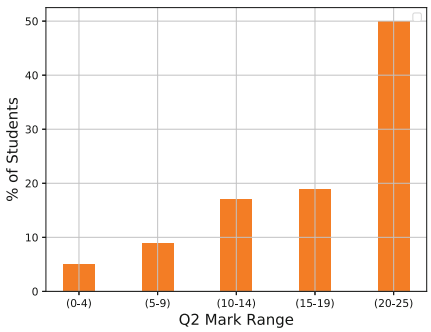
The final exam is 2-h long and pen & paper based. For full marks, students must correctly answer three out of four questions on the paper. In the case that a student answers all four questions, the best three are combined for their final grade. The overall exam structure is outlined in Table 1. Each question is weighted equally (25 marks) and focuses on examining a different topic. For example, Q2 in Table 1 is designed to examine the basic knowledge of two algorithms: Tseitin transformation [19] and DPLL [6], while Q3 is designed to examine Hoare Logic [11].

3.2 Exam Results

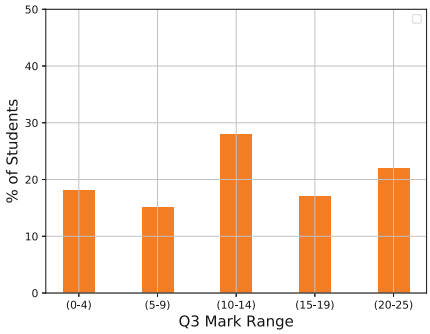
Overall, a total of 92 students, during the 2018–2019 academic year, participated in the module. In total, 23 of them failed resulting in a 25% failure rate. We



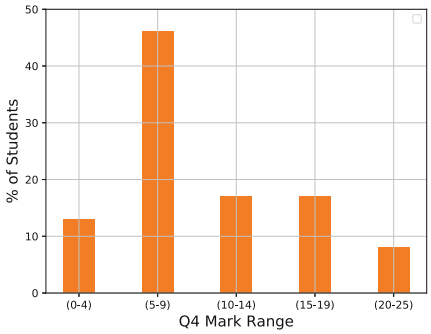
(a) Mark distribution for Question 1.



(b) Mark distribution for Question 2.



(c) Mark distribution for Question 3.



(d) Mark distribution for Question 4.

Fig. 1. Marks distribution for four questions in Table 1.

analyse the exam results on a per question basis as illustrated in Fig. 1. For each of these graphs, we plot the mark range (x-axis) against the percentage of students that answered this particular question in the exam (y-axis).

We can see from Fig. 1 that the students performed the best on Q2. In fact, Q2 was the most popular question with 88 out of 92 students attempting it. We believe that this is due to the mechanical nature of Q2. Once a student masters applying the corresponding rules, he/she is able to solve the basic problems on the fly. Hence, Q2 was the most popular and highest scoring out of the four exam questions.

Conversely, Q4 was the least popular. Only 24 out of 92 students attempted this question. Q4 was designed to challenge students on the following topics:

- Encode a simple specification into SMT formulas.
- Write a Spec# program with the appropriate specifications corresponding to a simple C# function that computes the sum of an integer array.
- Show basic SAT-encodings for reachability, safety and liveness properties.

We believe that this question was the least popular because it requires students to understand low-level SAT/SMT encoding plus writing specifications for a piece of code. In general, this type of question was not appealing to the students and this was also observed by the tutors during the weekly laboratory assignments.

4 Reflecting on Teaching and Learning

In this section, we outline our own experience as both student and educator. We reflect on this experience in light of the above examination results and outline our observations. Based on these observations we develop two hypotheses in relation to making this module more accessible and enjoyable from the students' perspective.

4.1 Our Experience

We summarise our own experience of learning (from our time as students studying this module) and our experience of teaching this module below. We note that the content of this module has not changed significantly in the time that has passed since we originally studied it during our undergraduate degrees.

Learning:

- The content of this module is generally challenging for students. Particularly, identifying loop invariants, presenting Hoare logic proofs and understanding low-level SAT/SMT encodings.
- Verification tools in general are not very reliable and the online versions of the tools frequently stopped responding during the lab sessions. Furthermore, the feedback from the tools is normally not very helpful in terms of figuring out what to prove or where one went wrong in the specification.
- Practical applications of formal verification are not very clear to the students.

Teaching:

- It is very difficult to help the students to see the value in the module since lots of these techniques are not widely used in industry.
- Many tools are not scalable for real-world examples and this makes it difficult to demonstrate their usefulness to students.
- It is necessary to use a combination of slides and manually working through examples on the whiteboard to explain the detailed computation steps (e.g. Hoare Logic) to the students.

We studied this module ourselves some years ago as students, and upon reflection, we believe that the content of this module has always been quite challenging. Our experience of teaching this module (both as lab tutor and lecturer),

interacting with the students and students' feedback forms[2] have revealed that this perception has not changed. Therefore, to encourage students, it is necessary to make improvements to this module. In order to identify such improvements we first outline four observations and then form two hypotheses in the next subsections.

4.2 Observations

By combining the examination results (Sect. 3) and our own experiences above, we make the following four observations:

Observation 1. Automated verification tools are not appealing to the students even though we described real world disasters that could have been avoided by using formal methods [14]. The tools that we use in this module for automated reasoning are Spec# [1] and Z3 [7]. The students use the online versions of Spec#[3] and Z3[4] as part of their practical lab work. In the beginning, the students found the click-button and go style interesting. However, they subsequently discovered that the feedback from the tool was not usually helpful for fixing bugs in the source code. We believe that this makes the tools harder to use and causes the students to lose interest. This also explains the reason for so few students, 24 out of 92, attempting Q4 as described in Table 1.

Observation 2. Most of the students performed reasonably well on natural deduction proofs but not using the Coq interactive theorem prover [3]. For natural deduction proofs, most of the students have already studied the material from their discrete structures module. However, they feel that it is difficult to find connections between the proofs worked out on a piece of paper and the corresponding sequence of Coq commands although multiple live Coq proof sessions are given throughout the lectures.

Observation 3. The verification tools and techniques that we have developed during our research were not integrated into this module. As such, the students were not given the opportunity to learn about our work. A portion of our research focuses on using SAT/SMT solving techniques to tackle problems from software engineering domains which is particularly relevant for this module [21,22]. Unfortunately, we never had the chance to present our work due to time constraints.

[2] At the end of the semester, the university distributes feedback forms for the students to fill in for each module that they have taken. It is not compulsory for the students to complete them and they ask broad, non module specific questions. We received a small number of responses and have used these to inform our discussion but we rely more heavily on the exam results and our interactions with the students.

[3] https://rise4fun.com/SpecSharp.

[4] https://rise4fun.com/Z3.

Observation 4. In general, the students had mixed reactions to the Hoare logic part of the module. In particular, the worked out whiteboard examples showing how to discover loop invariants were difficult to digest for some students. This usually involved interactive sessions during the lectures where the lecturer and students worked together to solve the problems. Others found the whiteboard examples to be extremely helpful when studying Hoare Logic. These students typically had a strong background in mathematics. We speculate that these students were accustomed to whiteboard style teaching whereas pure computer science students were more likely familiar with electronic slides.

These four observations reveal a number of shortcomings for this module. In particular, Observations 1 and 2 point to a lack of tool usability. This is a challenge for the formal methods community at large and can also hinder the uptake of formal methods in industry. Observation 4 noted that the students generally found Hoare logic difficult to grasp but this may also be exacerbated by the students' difficulty in using tools such as Spec#.

As a result of Observation 3, we have already started to integrate our research tool into current teaching. For example, we introduced our own tool, MaxUSE [21,22][5], into one of the classes and showed the students how to use it to find conflicting class invariants for a UML class diagram. A number of students clearly showed interest in the tool and would like to know more about its underlying algorithms and theories.

Based on these observations, we derive two hypotheses for improving this module in the next subsection.

4.3 Hypotheses

In this subsection, we develop two hypotheses that are based on the observations derived in the previous subsection. We intend to use these hypotheses to guide future improvements to be made to this module that we plan to investigate during the current (2019–2020) academic year.

Hypothesis 1. The development of an online repository that contains a collection of real world examples would be useful for both teaching and illustrating industrial uses of formal methods to the students. These examples could be proved by either using automated or interactive verification tools such as Z3 and Coq. We believe that this would create a strong connection between the theory taught in the class and practical, real world applications. However, the challenge here is that the examples collected or manually created should be small, but detailed enough to be suitable for educational use. One way to begin is to design and distribute a survey among the past students in order to identify the most interesting and educational examples to be used in the class.

[5] https://github.com/classicwuhao/maxuse.

Hypothesis 2. A platform such as Tarski's world that turns different kinds of logical reasoning proofs into games would increase the interactions between lecturers and students [2]. Hence, we believe that this is a good way to attract students to formal methods. For example, a live coding session that works with the students using SMT solvers to solve a Sudoku puzzle would be much more enjoyable and interesting than simply elaborating on different SMT constructs in the slides. However, there are two primary challenges that arise from this: (1) it may not be possible for each student to bring a laptop to the lecture, and, (2) students who miss the pre-setup steps may break the pace of a lecture. One potential solution is for the lecturer to show the code (solving games) running on their own machine and to upload the source code after the lecture so that the students can try it in their own time. However, in this way the interactions between the lecturers and students might be significantly reduced.

We have derived these hypotheses from our own experiences and the observations that we have made. We intend to investigate these hypotheses as future work to see if they improve the student experience and the exam results.

5 Conclusions and Future Work

Teaching formal methods is quite challenging and making formal methods appealing to younger generations is very important for continuously expanding the formal methods community in both industry and academia. In this paper, we discuss our own experience of both studying and teaching the same software verification module at Maynooth University. Based on our experiences and analysis of the exam results from the 2018–2019 academic year, we have derived four key observations in Sect. 4.2, from which we construct two hypotheses (Sect. 4.3) that we will investigate during the current (2019–2020) academic year.

Furthermore, we plan to work with the education research group within the department (at Maynooth University) to design interesting experiments in order to figure out the best way of teaching formal methods and to let students have fun with it. These experiments include interviewing students about specific topics covered during the lectures, gathering and analysing real feedback from the current academic year and using game based strategies to teach students to use different verification tools [16]. We believe that these experiments can help us to encourage the students to use formal methods/verification tools in their careers after their university studies.

Since Maynooth University also offers a similar module at Master's level, we plan to investigate the corresponding exam results and compare them with those presented in this paper. Furthermore, a much more detailed student feedback form will be distributed at the end of the module for further analysis.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
2. Barwise, J., Etchemendy, J.: Tarski's World: Version 4.0 for Macintosh (Center for the Study of Language and Information - Lecture Notes). Center for the Study of Language and Information/SRI (1993)
3. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st edn. Springer (2010)
4. Cataño, N.: Teaching formal methods: lessons learnt from using event-B. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 212–227. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_14
5. Creuse, L., Dross, C., Garion, C., Hugues, J., Huguet, J.: Teaching deductive verification through Frama-C and SPARK for non computer scientists. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 23–36. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_2
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Dean, C.N., Hinchey, M.G.: Teaching and Learning Formal Methods. Morgan Kaufmann, San Francisco (1996)
9. Gallardo, M.M., Panizo, L.: Teaching formal methods: from software in the small to software in the large. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 97–110. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_7
10. Gibson, J.P., Méry, D.: Teaching formal methods: lessons to learn. In: IWFM. Citeseer (1998)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
12. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press (2004)
13. Jaume, M., Laurent, T.: Teaching formal methods and discrete mathematics. In: 1st Workshop on Formal Integrated Development Environment, vol. 149, pp. 30–43. EPTCS (2014)
14. Jazequel, J., Meyer, B.: Design by contract: the lessons of Ariane. Computer **30**(1), 129–130 (1997)
15. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall (1988)
16. Moller, F., O'Reilly, L.: Teaching discrete mathematics to computer science students. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 150–164. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_10
17. Oliveira, J.N.: A survey of formal methods courses in European higher education. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 235–248. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30472-2_16
18. Rozier, K.Y.: On teaching applied formal methods in aerospace engineering. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 111–131. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_8

19. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Stud. Math. Math. Logic **2**, 115–125 (1968)
20. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surveys (CSUR) **41**(4), 19 (2009)
21. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 179–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_11
22. Wu, H.: MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 348–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_23