Teaching Formal Methods: A Systematic Survey

Simon Foster*
Department of Computer Science, University of York

30th March 2023

Abstract

Formal methods is an important branch of software engineering that is difficult to teach well. Effective teaching must therefore be informated by the latest pedagogical research. In this article, we perform a systematic survey of formal methods teaching literature over the past 10 years. We consider the curriculum, teaching techniques, software tools, and past course experience. We draw a number of lessons to be learned, that will inform our pedagogical approach for teaching formal methods in the second and third year computer science programme.

Keywords: formal methods teaching; software engineering; logic; proof; verification

1 Introduction

Formal methods is a branch of software engineering (SE) that focusses on mathematical analysis techniques that ensure software of high quality. Though it has a proven track record in improving computerised systems, it is notoriously difficult to teach well. Students can often be puzzled by the complex mathematical notation involved, and wonder what the applications are, to name but two issues.

The author is a lecturer in the department of computer science. He has previously taught a third-year optional formal methods course called "Assurance and Proof" (PROF). Though the course received excellent feedback and exam results were good, it was taught to a small cohort of around 15. Next year numbers will be at least 30, and therefore the course needs to scale. We are also delivering an industrial CPD based on this module in June, which will require adaptation for a different kind of audience. More significantly, Simon will also be teaching half of our mandatory second-year theory module, on logic and proof (THE3). This is a very different prospect, with a cohort of potentially 200 students. If these modules are to be successful, then it is vital that our teaching is informed by the latest pedagogical research.

The aim of this study is systematically review the pedagogical literature over the past ten years in formal methods research. There is a growing recognition that the challenge of teaching formal methods (FMs) should be recongised as a significant and unique area of pedagogical research. Over the past ten years, there has been a growing and increasingly vibrant community, demonstrated through events like the Formal Methods Teaching Workshop (FMTea), which happens biannually. The pedagogical literature is primarily about sharing experiences in teaching FMs, and to help other members of the community improve.

*Email: simon.foster@york.ac.uk

2 Research method

We performed a systematic review using Google Scholar and Scopus. We applied the search term "Formal Methods" AND ("Teaching" OR "Pedagogical"), and limited the time scale to the past 10 years (due to technological advances). We discovered that this missed out several important works from the "Innovation And Technology In Computer Science Education", and so we added "formal methods" literature from this venue. During the review, we discovered papers that do not self-identify as formal methods, which we added. We also corresponded with Colleagues from Korea and USA, which turned up some further papers in this category.

Our search yielded a total of 53 papers, which we then proceeded to read. Our chosen research question is *"What are the necessary preconditions to allow the use of formal methods in constructively supporting the computer science undergraduate curriculum?"*. An assumption of this is that FMs can be integrated and serve computer science, which is confirmed by our review.

The quality and nature of the papers is very variable. Many are discursive and anecdotal, based on teaching experience. Their origin is global, including authors from Europe, USA, Iceland, Australia and Russia. A few have quantitative data, but often the sample size is small and comparison is difficult. As a result, we instead approached answering our research question through deep engangement with the authors arguments, emphasising themes where there is clear consensus of opinion. Where convincing quantitative data is available, we emphasise this in the text, and generally prioritise the arguments of these papers.

The literature indicates that there are substantial lessons to be learned. In order to guide our discussion, we have divided our the literature review into 4 key subsections: (1) Curriculum and Module Design; (2) Teaching Techniques; (3) Software Tools; and (4) Course Experience. We then collate a set of lessons learned, and explain how these feed into our pedagogical practice.

3 Literature review

3.1 Curriculum and module design

In this section, we consider what an FM module should look like, how it should fit into the rest of the curriculum, and how it should be motivated.

Drachova et al. (2015) argue that high quality software requires teaching FMs. Cowling (2015) argues that SE is only "engineering" if discipline and rigour are applied, including FMs. Moller and O'Reilly (2019) argues that SE is unusual from other engineering disciplines, in that discrete mathematics are not taught well, which contributes to the "software crisis". FMs are often detached from the curriculum, and can scare students. They advocate for improved pedagogical techniques that instil rigour and formality in the early stages of the curriculum.

Favre (2018) along with Khazeev et al. (2019) argue that FMs are applicable across SE, and are not just safety-critical systems. Chaudhari and Damani (2015) and Ölveczky (2021) argue that FMs should be seamlessly integrated into the curriculum. Askarpour and Bersani (2019) argue that teaching computational thinking at school-level is needed to improve undergraduate FM teaching. Curzon, Bell, et al. (2019) argue that formal reasoning about algorithms is core to "computational thinking". Gallardo and Panizo (2019) argues there is a vicious circle in FMs not being taught well, if at all, and

so not being used. Yatapanage (2021) concurs, and supports this with their experience in teaching a second-year concurrency course.

Cerone and Lermer (2019) investigate approaches to engaging different target audiences in FM teaching. Ölveczky (2021) argues that courses must show the relevance and power of FMs in industrial practice. N. Jeppu, Y. Jeppu, and Devi (2017) stresses the importance of industrial involvement, and Gallardo and Panizo (2019) recommends use of significant case studies. Nair, Y. Jeppu, and Tahiliani (2020) advocate industrially applied techniques for rigorous engineering. They all argue for motivation with accidents where FMs would have helped (ExoMars 2016, Therac 25, Ariane 5 etc.), and FM success stories (Amazon Web services, 5G standard, VISA).

Drachova et al. (2015) present a comprehensive set of learning outcomes to guide the coherent integration of FMs into the curriculum. The topics have been taught at sophomore-level software foundations courses, and have been adopted by 11 Universities. Zamansky and Farchi (2015) conduct a small empirical study of their course, "Logic and Formal Specification", with a survey on perceptions on the usefulness of FMs and course content. The results indicate that teaching FMs improves students ability to undertake "computational thinking".

In summary, the literature indicates that FMs should be integrated coherently into the curriculum and motivated with real-world techniques and examples. However, there remain barriers to how this can be achieved, and so more pedagogical research is needed, which we consider in the next section.

3.2 Teaching techniques

In this section, we consider successful pedagogical techniques for FMs.

Cerone, Roggenbach, et al. (2011) emphasise the use of software tools and modelling examples in lab classes. There should be more emphasis in the *method* for constructing models, rather than mathematics. Teaching should focus on crucial paradigms, to allow depth of learning in specific situations (cf. Cowling (2015)). Knobelsdorf et al. (2017) argue that learning barriers stem from difficulties in mastering mathematical language, and guidance is needed, again, in the method of creating proofs.

Yatapanage (2021) emphasise teaching basic concepts, such as program state, in detail and with examples. Technical details should be minimised, until students have gained practical experience (Aceto and Ingólfdóttir 2021). Sekerinski (2019) and Gallardo and Panizo (2019) similarly argue for thorough explanation of notation. Natural language and graphics can lower the bar of entry. Cerone and Lermer (2019) argues that graphics can be used to illustrate concepts, but should be linked with mathematics to support reasoning. FM should be "fun", with engaging examples such as puzzles.

Ölveczky (2021) argues that key topics (modelling, requirements, reasoning, etc.) should be repeatedly taught (spiral model). He concurs with Cerone, Roggenbach, et al. (2011) that a small number of FMs should be taught, and argues for a single, expressive, and executable formalism that can be applied to a range of realistic examples. Tools should provide a high-level of automation, and be accessible to those without a mathematical background. Bella (2019) argues for the use of well-known examples, such as physical laws, to show that students have already used FMs "without knowing it". This helps to overcome psychological barriers, and allows students to have a deeper understanding.

Moller and O'Reilly (2019) present an innovative approach to teaching discrete mathematics in the first year. They emphasise (1) use of careful English in teaching

logic, and introduction of formality as a "shorthand" (method rather than mathematics); (2) focus on modelling of computer systems to engage students; (3) use puzzles and games to instil the rigours of "computational thinking"; (4) use of PBL. This dramatically increased the students getting a first, and reduced the failure rate.

Similarly, Curzon and McOwan (2013) demonstrate how magic shows have been used to teach FMs to school children and professionals. They situate this within the "computer science unplugged" agenda, which promotes constructivist learning through physical engagement in an activity. They present a selection of card tricks that produce a guaranteed outcome (the "magic"), and use proof to justify the answer. This work was later extended by Ferreira and Mendes 2014 to teach first year undergraduates concepts like specification and Hoare logic. Their approach includes execution of a card trick in the classroom, followed by PBL to develop the specification and algorithm. This allowed the students to take ownership, and derive their own unique solutions.

In addition to puzzles, several authors, including Ölveczky (2021) argues, advocate use of programming itself as a teaching aid. Chaudhari and Damani (2015) argue that teaching program verification should be taught alongside programming. Yatapanage (2021) support teaching with "simple" example programs, which nevertheless exhibit unexpected behaviour, which motivates formal analysis.

VanDrunnen (2011) and Jaume and Laurent (2014), argue for the teaching of *functional programming*, which is higher level and closer to discrete mathematics. Jaume and Laurent (2014) take this one step further by providing a tool to support teaching logic and discrete mathematics, which provides immediate feedback for user exercises. Farrugia-Roberts, Jeffries, and Søndergaard (2022) agrees that programming can itself be a vehicle for learning. Mathematical concepts can be embedded in the functional programming language Haskell, and exercises framed as programming problems.

Fleischmann et al. (2019) report on teaching a heterogeneous audience. The teach all the material twice (spiral model), first at a high-level with introductory examples, and second in detail with formal proofs. They use PBL, where students discuss exercises, supervised by a doctoral student. They have frequent testing with four evenly-spread one-hour exams, with two attempts for each. These features have led to a successive increase in the pass rate over 3 course iterations, with a growing cohort.

3.3 Software tools

In this section, we survey software tools that are used in teaching. We determine the capabilities and weaknesses of the tools, and set the context for the section on course experience. We divide the tools into four broad categories: proof assistants, program verifiers, model exploration, and model-based engineering.

3.3.1 Proof assistants

A proof assistant is a tool to support users in creation of mathematical proofs. Key examples are Isabelle and Coq, which have both been applied in an industrial setting.

Nipkow (2012) reports on teaching using Isabelle. He highlights its value over pen-and-paper proofs, due to instant feedback. Proof assistants are like video games, where winning corresponds to completion of a proof. This avoids students producing non-sensical proofs and promotes deep learning. Also, Isabelle/Isar provides a notation similar to pen-and-paper proofs, which helps overcome the learning gap with textbooks. Isabelle allows users to execute programs, and is therefore is suited for

SE. However, Nipkow advocates proof assistants as an aid to teaching, rather than as something to focus teaching on.

Villadsen and Jacobsen (2021) reports on teaching with Isabelle on two logic courses using "Mathematical Logic for Computer Science" by Ben-Ari as the course textbook. Their course is different to that of Nipkow (2012), since its emphasis is on proof systems. They report usability problems with Isabelle for the students, particularly with feedback. They have therefore developed their own front-end interfaces for Isabelle, called Sequent Calculus Verifier (SeCaV) and Natural Deduction Assistant (NaDeA) for teaching.

Knobelsdorf et al. (2017) perform an experiment using Coq as a learning tool. Their method was to deliver the course, use surveys, and to observe how students were approaching the exercises. No serious usability problems were observed and students were able to transfer their programming knowledge. Students displayed more satisfaction in developing proof with Coq, due to the immediate feedback. Whilst pen-and-paper proofs were very difficult, as knowledge of Coq increased so creating proofs became easier. They conclude that, with carefully crafted exercises and a usable interface, Coq can be an effective learning tool.

Loos and Platzer (2014) (cf. Platzer (2013)) report on teaching cyber-physical systems with the KeYmaera proof assistant. Its use allows more realistic models to be created, and allows students to gain a deeper understanding. Moreover, KeYmaera provides substantial guidance on proof, which avoids the "empty page" syndrome.

Leach-Krouse (2018) describes Carnap, an educational proof assistant with a web-based interface. It has been used for several years to support formal logic teaching at Kansas State University, and around 30 other institutions. Compared to general purpose tools like Isabelle and Coq, Carnap lowers the bar of entry for students, whilst retaining instant feedback.

3.3.2 Program verifiers

Program verifiers are targeted at verifying properties of imperative code. Although similar to proof assistants, they are more tailored to the task, and may provide a higher degree of automation. Usually, such tools are based on Hoare logic, which is a standard logic for program verification.

Blazy (2019) describes the use of the Why3 verification platform in a third year undergraduate course at the University of Rennes to a cohort of 100 students. The Why3 tool supports verification of imperative programs and is highly automated. Students write specifications, test them with execution, and finally verify them using invariants. The authors are positive about the use of Why3, because it provides very good low-level feedback on code issues.

Noble et al. (2022) reports their pedagogical approach for teaching program verification supported by the Dafny tool. They argue for a top-down (spiral) approach, where high-level program verification is taught first, and then foundational concepts are introduced later. They use Dafny, because it integrates with C♯ and Java, languages with which students are familiar. They highlight the value of Dafny's automated feedback, which is useful in formative assessment, and a Web interface.

Bubel and Hähnle (2016) argue that tools need to support reasoning in the same form used by standard computer science textbooks (e.g. Huth and Ryan (2004)), which tools like Why3 and Dafny often do not. Moreover, such tools provide high level of automation, which may allow students to complete assignments by trial-and-error, rather than by deep learning. Bubel and Hähnle (2016) therefore tailor Hoare logic for ease

of teaching, and implement this logic in a tool called KeY-Hoare. Sznuk and Schubert (2014) similarly argue that teaching should be accompanied by attractive, practical tools, and develop Hoare Advanced Homework Assistant (HAHA). HAHA was likewise developed to support teaching using traditional learning materials, and is shown to improve assignment scores compared with traditional pen-and-paper methods.

Divasoón and Romero (2019) evaluate program verification tools to support teaching. They reject proof assistants like Coq and Isabelle due to the steep learning curve. They reject Dafny and HAHA, as they prefer to focus on the Java language, due to student experience. They consider three tools in-depth: Krakatoa (based on Why3), KeY and OpenJML. They choose Krakatoa, since it can handle all their exercises. The authors quantitatively demonstrate an improvement in pass rates in their course since Krakatoa was introduced in 2013. Moreover, they argue that the use of the tool promotes deep learning for the students.

From a more critical standpoint, Farrell and Wu (2019) reports on issues encountered with tools like Coq and Spec# (similar to Dafny). The tools were not reliable, did not scale to realistic examples, and do not give detailed feedback. They do not appear practical as they are not widely applied in industry. Students could not make the link between pen-and-paper proofs and those in Coq. They propose two hypothetical solutions to these problems: (1) creation of an online repository of realistic examples; (2) use of a unified teaching platform like "Tarski's world".

3.3.3 Other tools

Several authors have applied model exploration tools in teaching. Aceto and Ingólfdóttir (2021) applies the model checker Upaal, and Dubois, Prevosto, and Burel (2019) have applied SPIN. Ölveczky (2021) uses the rewriting system, Maude, in his second year undergraduate module. Korečko and Sorád (2015) show how train-based simulation games can be an effective aid to learning. Space will not allow further detail.

In a different axis, several authors have proposed the use of model-based engineering (MBE) tools, following arguments given by Cowling (2015) and others. Favre (2018) proposes a modelling language called Nereus, as an interface to other verification tools. Lipaczewski and Ortmeier (2013) similarly provides an intermediate language, called "Safety Analysis Modelling Language" (SAML), which supports formal analysis of higher level MDE languages. Fisher and Johnson (2016) argues that it is insufficient simply to demonstrate the potential benefits of FMs, but to show that they can be applied broadly, and advocate automated test generation.

We conclude this section with some observations. Learning can certainly be aided by software tools, but usability and feedback quality are essential. There is also a balance to be struck between automation and pedagogy. As with arithmetic and electronic calculators, students needs to understand both the theory and its application.

3.4 Course experience

In this final section of the literature review, we consider FM course experience.

Farrugia-Roberts, Jeffries, and Søndergaard (2022) use Haskell in the "Grok Academy" platform to teach a large cohort of 500 undergraduates. Exercises require students to solve a specified problem, such as "construct a deterministic finite automaton", and the interface checks the answers. They scaffold with partially complete functions, hands-on open development, and rapid contextualised feedback to students in a uni-

fied tool. With a focus on programming, it is more attractive to CS students. A barrier is potential unfamiliarity with Haskell, which is less common than other languages, though they argue that benefits outweigh the cost of learning it.

Aceto and Ingólfdóttir (2021) describes an intense three week undergraduate course in applied FMs at Reykjavik University, which applies Upaal. The course is project-based and student driven, with broad exercises. In the first week, they learn the tool by modelling puzzles and examples[1]. In the second week they model a robot vacuum cleaner, and each student delivers a report. In the final week, the students explore different ways to solve the puzzle game "Rush Hour". Each solution is presented as a group, and the lecturers ask questions. The course has had consistently high feedback, with student satisfaction averaging at 4.79/5 over 8 years.

Greenberg and Osborn (2019) report on using the Coq-based "Software Foundations"[2] materials to teach discrete mathematics to first- and second-year undergraduates. Their teaching technique interleaves formal mathematics in Coq with informal presentations on the white board. They report that the Software Foundations materials required substantial adaptation, and the students found it difficult to use Coq.

Cataño (2017) reports on their course using the Event-B method, which they argue is more suited to software engineers. Based on this experience, they recommend the use of a large collection of modelling examples; using proof assistants (like Coq and Isabelle) to support teaching with user feedback; and using Event-B to support modelling. Cataño (2019) takes this further and reports on a later version of the course involving formation of software development teams.

Dubois, Prevosto, and Burel (2019) reports on an SE programme with a FM track, spread over several modules, which has run for 15 years. FM tools used include the SPIN, Event-B, and Coq. The latter are not used for exercises, due to the limited time students are exposed to them, and exercises are mainly pen-and-paper in nature.

Gallardo and Panizo (2019) report on teaching a compulsory FM course over five years to a cohort of around 50. They employ the spiral model, with different tools applied to different aspects of the same models; teaching thus emphasises strengths and weaknesses of them. The song "I'm my own grandpa" is used as an example of natural language than can be formally modelled.

Khazeev et al. (2019) reports on using the Eiffel language at Innopolis University in Russia. Their exercises are partial programs with blanks to be filled in. Though students were positive, there was a perception that the techniques would only benefit the very small number of Eiffel users. They stress the need to show how techniques generalise to everyday systems.

Rozier (2019) report on an applied FM course that is part of an industrial Aerospace Engineering programme at Iowa State. The first part is a broad survey of FM techniques (e.g. SPIN, Coq, and Isabelle), with group exercises (and prizes). The second part is a group project where the students apply one of the tools to a modelling task, with self-defined success criteria. Students therefore gain knowledge of a wide range of techniques used by industry, though some this knowledge may be shallow and students struggle with some of tools, particularly the proofs assistants.

Simpson (2019) reports on a one-week intensive masters-level course at Oxford. It teaches FMs and discrete mathematics using the Z notation. The course is accompanied by the textbook *Using Z* by Woodcock and Davies. Parallels are drawn with the course and concepts described by Jaume and Laurent (2014). The authors de-

[1] Upaal Examples: `https://homes.cs.aau.dk/~kgl/ESV04/exercises/index.html`

[2] Software Foundations: `https://softwarefoundations.cis.upenn.edu/`

scribes a recent shift in approach, from individual exercises to group-based PBL. Like Cowling (2015), they focus on the method of developing and verifying models. Having learned the theoretical topics early in the week, a whole day is spent on group modelling exercises. Here, the students are given an English language description, and a part completed Z specification with blanks to be filled in. This change resulted in an improvement in course satisfaction.

4 Discussion

We conclude with outputs that feed into our teaching plan for THE3 and PROF. FMs should be carefully integrated into the curriculum, and not treated as a separate "magic craft" only for the suitably initiated. Indeed, students should be encouraged to see they have used FMs without knowing it, to improve their understanding and confidence. We will therefore motivate with real-world examples, ideally with simpler versions being projected into the curriculum to demonstrate how techniques transfer. We will also endeavour to have one of our industrial research partners give a guest lecture.

Examples and exercises should be accessible and engaging, with logic puzzles being particularly relevant. These can be used to support a spiral model, where the "big idea" is shown first, and then the theoretical details explained in additional steps. Construction of models, proofs, and other artefacts should focus on the *method* rather than mathematics. When teaching logic, structured natural language should be used first, and then mathematical notation to abbreviate and support reasoning. Moreover, a concrete output of this review is a curated set of illustrative examples (e.g. Quiz Show, Knights and Knaves, Wolf-Goat-Cabbage) that have previously been successful used in the aformentioned courses, and will be used in both THE3 and PROF. Exercises should be scaffolded by providing a partially completed answer.

Software tools are indeed valuable, but care is needed in ensuring they provide adequate feedback. We should minimise the number of tools being taught, but ensure the ones used have demonstrable applications. For THE3, we intend to use two tools: Carnap and Isabelle. Carnap is simple, user-friendly, web-based, and stable, and so is ideal for teaching second years. However, it is purely a teaching tool, and so we will use it as an "on-ramp" for Isabelle, which is more powerful, general, and has industrial applications. Isabelle can be applied to both modelling and programming (including execution and simulation), and can scale to realistic verification problems. It has many of the same advantages as functional languages like Haskell, and can be used to illustrate mathematical concepts. Nevertheless, we recognise quality of feedback in Isabelle remains a problem, and we are investigating better error reporting.

In spite of technological advances, textbooks are still popular and valued. A good textbook should match the criteria enumerated above in its presentation. We need to ensure our tools can support learning with them, by using the same format of mathematics and proof. Our chosen course text, "Logic in Computer Science" (Huth and Ryan 2004), uses a "Fitch-style" proof, which is supported in Carnap, which should help overcome student learning barriers. We agree that program verification tools need to balance faithfulness to the underlying theory with sufficient automation to ensure scalability. Isabelle is a unified tool that can be tailored to support learning activities at different levels of abstraction, and it can act a "gateway" to a variety of other FM tools. For example, we can both express low-level Hoare logic in the form used by textbooks (cf. KeY-Hoare, HAHA), and also high-level automation (cf. Why3, Dafny), as we've previously demonstrated in PROF.

References

Aceto, L. and A. Ingólfdóttir (2021). “Introducing formal methods to first-year students in three intensive weeks”. In: *Formal Methods Teaching (FMTea 2021)*. Vol. 13122. LNCS. Springer.

Askarpour, M. and M. M. Bersani (2019). “Teaching Formal Methods: An Experience Report”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Bella, G. (2019). “You Already Used Formal Methods but Did Not Know It”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Blazy, S. (2019). “Teaching Deductive Verification in Why3 to Undergraduate Students”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Bubel, R. and R. Hähnle (2016). “KeY-Hoare”. In: *Deductive Software Verification*. Ed. by W. Ahrendt et al. Vol. 10001. LNCS. Springer, pp. 571–589.

Cataño, Néstor (2017). “An Empirical Study on Teaching Formal Methods to Millennials”. In: *2017 IEEE/ACM 1st International Workshop on Software Engineering Curricula for Millennials*. IEEE. DOI: 10.1109/SECM.2017.1.

— (2019). “Teaching Formal Methods: Lessons Learnt from Using Event-B”. In: *Formal Methods Teaching (FMTea 2019)*. Ed. by B. Dongol, L. Petre, and G. Smith. Vol. 11758. Springer, pp. 212–227.

Cerone, A. and L. R. Lermer (Dec. 2019). “Adapting to Different Types of Target Audience in Teaching Formal Methods”. In: *Proc. 1st Intl. Workshop on Formal Methods – Fun for Everybody*. Vol. 1301. CCIS. Springer.

Cerone, A., M. Roggenbach, et al. (2011). “Teaching Formal Methods for Software Engineering: Ten Principles”. In: *Informatica Didactica* 9.

Chaudhari, D. L. and O. Damani (July 2015). “Introducing Formal Methods via Program Derivation”. In: *Proc. 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2015)*. ACM.

Cowling, A. J. (2015). “The role of modelling in teaching formal methods for software engineering”. In: *Proc. 1st Workshop on Formal Methods in Software Engineering Education and Training (FMSEE&T)*. Ed. by A. Bollin, T. Margaria, and I. Perseil. Vol. 1385. CEUR Workshop Proceedings.

Curzon, P., T. Bell, et al. (2019). “Computational Thinking”. In: *The Cambridge Handbook of Computing Education Research*. Ed. by S. A. Fincher and A. V. Robins. Cambridge University Press.

Curzon, P. and P. McOwan (July 2013). “Teaching formal methods using magic tricks”. In: *Proc. “Fun with formal methods” at 25th Intl. Conf. on Computer Aided Verification (CAV)*.

Divasoón, J. and A. Romero (2019). “Using Krakatoa for Teaching Formal Verification of Java Programs”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer, pp. 212–227.

Drachova, S. V. et al. (Aug. 2015). “Teaching Mathematical Reasoning Principles for Software Correctness and Its Assessment”. In: *ACM Trans. Comput. Educ.* 15.3.

Dubois, C., V. Prevosto, and G. Burel (2019). “Teaching Formal Methods to Future Engineers”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer, pp. 212–227.

Farrell, M. and H. Wu (2019). “When the Student becomes the Teacher”. In: *Proc. Intl. Workshop on Formal Methods – Fun for Everybody*. Vol. 1301. CCIS. Springer, pp. 208–217.

Farrugia-Roberts, Matthew, Bryn Jeffries, and Harald Søndergaard (2022). “Programming to Learn: Logic and Computation from a Programming Perspective”. In: *Pro-*

ceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1. New York, NY, USA: ACM, pp. 311–317. ISBN: 9781450392013.

Favre, L. (2018). "Teaching Formal Methods in the Context of Model Driven Engineering". In: *EDSIG Conference on Information Systems and Computing Education*. ISCAP.

Ferreira, J. and A. Mendes (June 2014). "The magic of algorithm design and analysis: teaching algorithmic skills using magic card tricks". In: *Proc. 2014 conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, pp. 75–80. DOI: 10.1145/2591708.2591745.

Fisher, G. and C. Johnson (2016). "Making Formal Methods More Relevant to Software Engineering Students via Automated Test Generation". In: *Proc. 2016 conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, pp. 224–229. DOI: 10.1145/2899415.2899424.

Fleischmann, P. et al. (2019). "Managing Heterogeneity and Bridging the Gap in Teaching Formal Methods". In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Gallardo, M. and L. Panizo (2019). "Teaching Formal Methods: From Software in the Small to Software in the Large". In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Greenberg, M. and J. C. Osborn (Jan. 2019). "Teaching Discrete Mathematics to Early Undergraduates with Software Foundations". In: *Proc. 5th Intl. Workshop on Coq for Programming Languages (CoqPL 2019)*.

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

Jaume, M. and T. Laurent (2014). "Teaching Formal Methods and Discrete Mathematics". In: *Proc. Formal Integrated Development Environment 2014 (F-IDE 2014)*. Vol. 149. EPTCS.

Jeppu, N., Y. Jeppu, and M. Devi (2017). "Teaching formal methods at undergraduate/graduate level: The three perspectives". In: *3rd Intl. Conf. on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pp. 310–315. DOI: 10.1109/ICATCCT.2017.8389153.

Khazeev, M. et al. (2019). "Reflections on Teaching Formal Methods for Software Development in Higher Education". In: *1st Intl. Workshop on Frontiers in Software Engineering Education*. Vol. 12271. LNCS. Springer.

Knobelsdorf, M. et al. (2017). "Theorem Provers as a Learning Tool in Theory of Computation". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. New York, NY, USA: ACM, pp. 83–92. ISBN: 9781450349680. DOI: 10.1145/3105726.3106184. URL: https://doi.org/10.1145/3105726.3106184.

Korečko, Štefan and Ján Sorád (2015). "Using simulation games in teaching formal methods for software development". In: *Innovative Teaching Strategies and New Learning Paradigms in Computer Programming*. IGI Global, pp. 106–130.

Leach-Krouse, Graham (2018). "Carnap: An Open Framework for Formal Reasoning in the Browser". In: *Proc. 6th International Workshop on Theorem proving components for Educational software*. Ed. by P. Quaresma and W. Neuper. Vol. 267. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 70–88. DOI: 10.4204/EPTCS.267.5.

Lipaczewski, M. and F. Ortmeier (2013). "Teaching and Training Formal Methods for Safety Critical Systems". In: *Proc. 39th Euromicro Conference Series on Software Engineering and Advanced Applications*. IEEE.

Loos, S. M. and A. Platzer (2014). “Teaching Cyber-Physical Systems with Logic”. http://lfcps.org/pub/TeachCPS.pdf.

Macedo, N. et al. (2020). “Experiences on Teaching Alloy with an Automated Assessment Platform”. In: *ABZ 2020*. Vol. 12071. LNCS. Springer, pp. 61–77.

Moller, F. and L. O’Reilly (2019). “Teaching Discrete Mathematics to Computer Science Students”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Nair, G., Y. Jeppu, and M. Tahiliani (2020). “Teaching EARS to Undergrads in the Pandemic – Industry Academia Experience”. In: *2020 IEEE Bombay Section Signature Conference (IBSSC)*. IEEE. DOI: 10.1109/IBSSC51096.2020.9332163.

Nipkow, T. (2012). “Teaching Semantics with a Proof Assistant: No more LSD Trip Proofs”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*. Vol. 7148. LNCS. Springer.

Noble, J. et al. (2022). “More Programming Than Programming: Teaching Formal Methods in a Software Engineering Programme”. In: *NASA Formal Methods (NFM 2022)*. Vol. 13260. LNCS. Springer, pp. 431–450. DOI: 10.1007/978-3-031-06773-0_23.

Ölveczky, Peter Csaba (2021). “Teaching Formal Methods for Fun Using Maude”. In: *Formal Methods – Fun for Everybody*. Ed. by Antonio Cerone and Markus Roggenbach. Vol. 1301. CCIS. Springer, pp. 58–91.

Platzer, A. (2013). “Teaching CPS foundations with contracts”. In: *Proc. 1st Workshop on Cyber-Physical Systems Education (CPS-Ed 2013)*. https://cps-vo.org/group/edu/workshop/proceedings2013, pp. 7–10.

Rozier, K. Y. (2019). “On Teaching Applied Formal Methods in Aerospace Engineering”. In: *Formal Methods Teaching (FMTea 2019)*. Ed. by B. Dongol, L. Petre, and G. Smith. Vol. 11758. LNCS. Springer.

Sekerinski, E. (2019). “Teaching Concurrency with the Disappearing Formal Method”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Simpson, A. (2019). “Teaching Introductory Formal Methods and Discrete Mathematics to Software Engineers: Reflections on a Modelling-Focussed Approach”. In: *Formal Methods Teaching (FMTea 2019)*. Vol. 11758. Springer.

Sznuk, T. and A. Schubert (2014). “Tool Support for Teaching Hoare Logic”. In: *Software Engineering and Formal Methods*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Cham: Springer International Publishing, pp. 332–346. ISBN: 978-3-319-10431-7.

VanDrunnen, T. (2011). “The case for teaching functional programming in discrete math”. In: *Proc. ACM Intl. conference on Object oriented programming systems languages and applications (OOPSLA 2011)*, pp. 81–86.

Villadsen, J. and F. Jacobsen (2021). “Using Isabelle in Two Courses on Logic and Automated Reasoning”. In: *Formal Methods Teaching (FMTea 2021)*. Vol. 13122. LNCS. Springer.

Yatapanage, N. (2021). “Introducing Formal Methods to Students Who Hate Maths and Struggle with Programming”. In: *Formal Methods Teaching (FMTea 2021)*. Vol. 13122. LNCS. Springer.

Zamansky, A. and E. Farchi (2015). “Exploring the Role of Logic and Formal Methods in Information Systems Education”. In: *Software Engineering and Formal Methods (SEFM 2015)*. Vol. 9509. LNCS. Springer. DOI: 10.1007/978-3-662-49224-6_7.