Rooting Formal Methods Within Higher Education Curricula for Computer Science and Software Engineering — A White Paper —

Antonio Cerone[1](✉), Markus Roggenbach[2], James Davenport[3], Casey Denner[2], Marie Farrell[4], Magne Haveraaen[5], Faron Moller[2], Philipp Körner[6], Sebastian Krings[7], Peter Csaba Ölveczky[8], Bernd-Holger Schlingloff[9], Nikolay Shilov[10], and Rustam Zhumagambetov[1]

[1] Nazarbayev University, Nur-Sultan, Kazakhstan
antonio.cerone@nu.edu.kz
[2] Swansea University, Swansea, UK
m.roggenbach@swansea.ac.uk
[3] University of Bath, Bath, UK
[4] University of Manchester, Manchester, UK
[5] University of Bergen, Bergen, Norway
[6] Heinrich-Heine-Universität, Düsseldorf, Germany
[7] Niederrhein University of Applied Sciences, Krefeld, Germany
[8] University of Oslo, Oslo, Norway
[9] Humboldt-Universität zu Berlin, Berlin, Germany
[10] Innopolis University, Kazan, Russia

Abstract. This white paper argues that formal methods need to be better rooted in higher education curricula for computer science and software engineering programmes of study. To this end, it advocates

- improved teaching of formal methods;
- systematic highlighting of formal methods within existing, 'classical' computer science courses; and
- the inclusion of a compulsory formal methods course in computer science and software engineering curricula.

These recommendations are based on the observations that

- formal methods are an essential and cost-effective means to increase software quality; however
- computer science and software engineering programmes typically fail to provide adequate training in formal methods; and thus
- there is a lack of computer science graduates who are qualified to apply formal methods in industry.

This white paper is the result of a collective effort by authors and participants of the *1st International Workshop on Formal Methods – Fun for Everybody* which was held in Bergen, Norway, 2–3 December 2019.

A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 1–26, 2021.
https://doi.org/10.1007/978-3-030-71374-4_1

As such, it represents insights based on learning and teaching computer science and software engineering (with or without formal methods) at various universities across Europe.

1 Introduction

The greatest contribution that universities make to industrial practices is through releasing legions of graduates every year. When properly equipped with a scholarly education, these graduates challenge established processes and pave the way for new approaches. In the increasingly-digital world we live in, the scope for this is arguably greatest in the software industry, particularly given that the public perception – and indeed the reality – is that software is inherently unreliable.

Advances in digital technology take place at an astronomical rate, unfettered by regulations which would hinder progress in other scientific endeavours. There are generally few established principles in place to ensure that new software systems are as reliable as, say, a new vaccine. Software engineers demonstrate success in their company by releasing systems which, for *almost* all intents and purposes, *appear* to work. Because of the benefits these advances offer society, the public are generally accepting of – and, indeed, used to – software failures.

This situation persists in spite of the fact that computer science and software engineering research has developed a multitude of design principles which could help to improve software quality [Bar11]. It has been over half a century since Robert Floyd's seminal paper [Flo67] set out the means by which computer programs could be analysed to determine their functional correctness, and formal methods for developing correct software have been steadily devised and refined ever since. The typical computer science or software engineering graduate, however, leaves university with little or no knowledge of formal methods, and even a dislike for whatever formal methods they have encountered in their studies. Thus, rather than opening doors for formal methods in (software) industry, university education seems to have a detrimental effect.

Due to their ubiquity, software failures are overlooked by society as they tend to result in nothing more serious than delays and frustrations. We accept as mere inconvenience when a software failure results in a delayed train or an out-of-order cash machine or a need to repeatedly enter details into a website. However, the problems of systems failures become more serious (costly, deadly, invasive) as automatic control systems find their way into virtually every aspect of our daily lives. This increasing reliance on computer systems makes it essential to develop and maintain software in which the possibility, and probability, of hazardous errors is minimised. Formal methods offer cost-efficient means to achieve the required high degree of software quality.

A major reason that students (and, in turn, software engineers) have a negative attitude towards formal methods is that these are not introduced with due care during the early stages of higher education. Left to the theoretical computer science professor, such courses often start with fearful terms like state machine, logical inference, mathematical semantics, etc., without providing elementary explanations of the basic notions which relate these to the practice of software development. In their defence, formal methods professors often find it difficult to deliver the subject due to students' scepticism [Zhu20], which arises from the generally limited or non-existent exposure to formal methods in the rest of the curriculum. Boute [Bou09] and Sekerinski [Sek06] observe that limited references from other subjects and isolated use are the main factors leading to students' low opinion. Even worse, students perceive formal methods to be unsuitable for actual software engineering [BDK+06] or even an "additional burden" [BLA+09].

In this white paper we analyse what hinders a successful formal methods education, and make constructive suggestions about how to change the situation. We are convinced that such changes are a prerequisite for formal methods to become widely accepted in industry. We analyse the current situation of formal methods teaching and explore ways which we think will be engaging for students and practitioners alike. Our vision is that formal methods can be taught in such a way that both students and lecturers will enjoy formal methods teaching.

This white paper is the result of a collective effort by authors and participants at the 1st International Workshop "Formal Methods – Fun for Everybody", which was held in Bergen, Norway, 2–3 December 2019. At the workshop, there were several discussion sessions. Based on these, the two lead authors devised a paper outline, which was subsequently "populated" with text snippets written by all authors. The resulting draft was carefully edited, and agreed upon by all authors. By its very nature, this white paper offers a spectrum of opinions, in particular in the personal statements. What unites us are the following beliefs:

- Current software engineering practices fail to deliver dependable software.
- Formal methods are capable of improving this situation, and are beneficial and cost-effective for mainstream software development.
- Education in formal methods is key to progress things.
- Education in formal methods needs to be transformed.

In Sect. 2, we analyse the challenges in teaching formal methods. In Sect. 3, we collect ideas about how to teach formal methods – the fun way. In Sect. 4, we discuss how to increase the visibility of formal methods throughout the curriculum. In Sect. 5, we suggest a syllabus for a compulsory formal methods course. Finally, we discuss how to assess such teaching efforts in Sect. 6, before making concluding remarks in Sect. 7.

2 Challenges in Teaching Formal Methods

Teaching of formal methods faces a number of challenges. Currently, as a knowledge area, formal methods are virtually absent from curricula in computer science or software engineering. Formal Methods barely appear in the ACM/IEEE 2014 Software Engineering Curriculum, and indeed the development of formal specifications is explicitly deemed to be inappropriate for a capstone project [ACM15, p. 56]. Moreover, many students have an incorrect perception of what formal methods are about. Formal methods neither make the headlines nor are a popular topic in social networks, nor are they visibly used by industry. It is also the case that colleagues as well as students have misguided ideas concerning the mathematical background required to utilise formal methods. In the following, we elaborate on these topics. The section concludes with personal statements.

We begin our discussion by providing a working definition, cf. [RCS+21], of what a formal method might be.

Definition 1. *A formal method M can be seen to consist of the three elements syntax, semantics, and method:*

- *Syntax: the precise description of the form of objects (strings or graphs) belonging to M.*
- *Semantics: the 'meaning' of the syntactic objects of M, in general by a mapping into some mathematical structure.*
- *Method: algorithmic ways of transforming syntactic objects, in order to gain some insight about them.*

A typical example of a formal method is the process algebra CSP: its syntax is given in form of a grammar; there are various formal semantics (operational, denotational, and axiomatic ones); and there are proof methods for refinement via model checking and theorem proving.

Applying this definition, e.g., to the programming language Pascal, we see that it also qualifies as a formal method. It has a defined syntax and formal semantics; and each compiler and static analyser provides a method, the Hoare calculus would be another instance of a method.

UML on the other hand does not qualify as a formal method. The syntax is largely fixed via meta models, and there are various methods available, e.g., for code generation (e.g., from class diagrams or state machines). However, proposed semantics for UML contain several critical "variation points" and has – to the best of our knowledge – never been fully formalised.

2.1 On the Absence of Formal Methods from Computer Science and Software Engineering Curricula

Anecdotal evidence suggests that current computer science and software engineering curricula rarely cover formal methods to a large extent. We exemplify this observation by providing an historic perspective on programming education, an element central to all curricula.

In the late 1980s, Pascal was a dominant teaching language for beginning programming students. Pascal is a small, structured programming language with a syntax designed to be easy to parse [ISO90]. Most textbooks of the time presented the Pascal language using syntax diagrams, alerting the students to the idea of context free grammars, e.g., [CC82]. The element of syntax was taught as an integral part of programming. Some textbooks included the entire ISO Pascal standard, thus making the students aware of language definition documents.[1] For those specifically interested, Pascal had a widely available formal semantics [HW73]. Robust programming, i.e., checking preconditions, was an essential part of programming courses. Some universities would even have space for a formal methods course, typically based on Hoare logic, in their undergraduate curriculum: i.e., a formal method was taught.

About 20 years ago, Pascal was superseded by Java as the dominating teaching language. Java is a much more complex language than Pascal; it supports object-oriented development, and it has large support libraries. Thus, in the transition to Java, precise syntax and semantics was replaced by a more example-driven approach, e.g., [DD07], where the first half contains similar material to [CC82]. Verification tools such as Java Pathfinder[2] rarely made it into the syllabus of a programming course. Instead, students needed to learn more methodology, such as object-orientation, test-driven design and agile methods. All of this reduces the students' exposure to formality, such as formal syntax or precise semantics[3], making the gap to formal methods larger. Further, the pragmatics of software development take up more of the curriculum, leaving less space for a formal methods course in the core curriculum. Dewar and Schonberg support this critical assessment: "It is our view that Computer Science education is neglecting basic skills, in particular in the areas of programming and formal methods. We consider that the general adoption of Java as a first programming language is in part responsible for this decline [DS18]."

In recent years, Python has emerged into a common teaching language for programming. The move towards Python represents a change back to a much smaller language than Java. The Python reference document is just 160 pages, and its formal grammar is only four pages [vRtPdt20]. This should make it possible to at least expose the students to formal syntax and a standardisation document. However, the typing and semantic model of Python remains complex, and is not easily formalised.

Thus, while current programming education based on Java often fails to provide foundations for formal methods by discussing syntax and semantics, the move towards Python provides the silver lining that the element of syntax might again become a part of standard education in programming.

---

1 Pattis [Pat94] even suggested teaching Extended Backus-Naur Form (EBNF) as the first topic in computer science. Not to facilitate presenting the syntax of a programming language, but because EBNF is a microcosm of programming. With no prerequisites, students are introduced to a variety of fundamental concepts in programming: formal systems, abstraction, control structures, equivalence of descriptions, the difference between syntax and semantics, and the relative power of recursion versus iteration.

2 https://github.com/javapathfinder/jpf-core/wiki.

3 The recent *The Java®Language Specification, Java SE 14 Edition* is 800 pages [GJS+20] and not easily digestible.

2.2 Students' Perception of Formal Methods

The reduced exposure to formal approaches, as described in Sect. 2.1, supports university students' misconception that formal methods are a difficult topic with little or no practical relevance. This keeps students away from formal methods during their undergraduate studies. Even worse, it leads them to embrace the common belief that mathematics and computer science are two independent, fully distinct disciplines. Computer science is rather identified with programming, which, in turn, is seen more like an art rather than a scientific activity [CL20]. Interestingly, this view has even been supported not only by the pragmatic evolution of programming languages outlined in the previous paragraphs, but also by some academic publications claiming that rigorous mathematical knowledge is not necessary for computer science practitioners [Gla00]. Finally, this view has been paradoxically encouraged by the introduction of computer science in high schools. In fact, although in several schools computer science has been introduced as a stand-alone subject, it is not connected with mathematics but, instead, it is presented as a 'service subject' intrinsically tied to the use of computers. Scope of the subject is to provide tools that facilitate students in carrying out their homework and class projects [Cer20, Gib08].

Although we can say that, on average, a typical computer science student tends to have a negative perception of formal methods, in reality lecturers observe a lot of variation between students, as well as changes of perceptions in one direction or the other. Variations in students can be observed starting from the first programming courses. A slightly exaggerated categorisation goes as follows. On the one hand, there are students who tackle programming in a purely 'artistic way' by sitting down at the computer and writing code immediately, using debugging rather than problem solving to reach the solution. On the other hand, there are students who start analysing the problem using pen and paper, then draw diagrams, possibly write pseudo-code, test their solution on paper and, only when they are confident in their solution, they sit in front of a computer and convert their solution into a program. Obviously, it is the latter approach what lectures suggest. Normally, the former group of students tend to have a negative perception of formal methods, whereas the latter group tend to have a positive one. This partition of the students in two groups appears more evident once recursion is introduced in the programming course. The former group of students will tend to hate recursion, the latter group will tend to love it.

These two opposite perceptions obviously occur in several degrees. Moreover, they are not static but, at least potentially, dynamic and may be either encouraged or hindered in various ways throughout the course of undergraduate studies. The common absence of formal semantics among the topics of programming courses definitely keeps students away from an early exposure to formal methods and prevents them from really understanding what formal methods are. Being exposed to some basic operational semantics could actually help students to better understand conditional and iterative constructs, which are normally serious challenges for first year students. Furthermore, recursion could be better understood, thus providing the basis for a future interest in formal methods.

Concerning senior students, although for some of them their perception of formal methods may have been strongly oriented towards the negative side, there is hope to shift them towards the positive side. Senior students tend to be very pragmatic and their minds are dominated by the goal of entering the job market and the industrial world. Therefore they will build a positive perception of formal methods when presented with their pragmatic and industry-oriented aspects.

2.3 Limited Visibility of Formal Methods in Media and Industry

How students perceive a knowledge area has many drivers, such as personal success, like/dislike of certain academic teachers, their grades, etc. But maybe 'coolness' is the dominant factor. During their studies, students want to do something cool, maybe work with AlphaZero[4] or participate in a hackathon such as Google's Hash Code. Students also strive to get 'cool jobs', e.g., with Google, Facebook, Amazon, and the like. Currently, what one might want to call the 'coolness factor' of formal methods is rather low. Formal methods make neither the headlines nor are prominent in social media, nor are they visibly used by industry.

Besides studying, quite a number of students work on the side for companies. In these jobs, students often see only small parts of the overall job profile of a professional computer scientist or software engineer. Many of these side jobs deal with having a quick and dirty solution for some pressing problem, adapting software according to customer requests, or building prototypes in order to find out whether some concept works out. In contrast, mature students, coming back from industry and getting into university education again, know about the importance of quality assurance. But as they usually were not exposed to formal methods in their jobs, they are often reluctant to study them.

Luckily, there is some serious uptake of formal methods in industry. The classic case of a safety-critical industry is railway signalling, as described e.g. in [GM13]. Ligne 14 of the Paris Métro had software built using the B method [GM13] and has now run for over 20 years without a bug being reported. The "High Integrity Systems" unit of Altran develops systems for, e.g., the railway signalling industry and air traffic control, as well as tools and methodologies, such as the SPARK subset of Ada [MC15]. SPARK 2014 uses contracts to describe the specification of components in a form that is suitable for both static and dynamic verification.

Outside the safety-critical industry, a few 'enlightened', large information technology companies are beginning to use formal methods:

- Google is developing an ecosystem for formal analysis tools [SvGJ+15].
- Facebook uses "advanced static analysis" as described in [DFLO19].
- Amazon's use of formal methods is discussed in [NRZ+15, BBC+19]. There is a more technical description of one component in [CCC+18].

[4] AlphaZero is the descendant of AlphaGo, the AI that became known for defeating Lee Sedol, the world's best Go player, in March of 2016.

If we look at Facebook, [DFLO19] reports that, in many cases, "we have gravitated toward a 'diff time' deployment, where analyzers participate as bots in code review, making automatic comments when an engineer submits a code modification". For their Infer tool, which has its origin in the separation logic work of [CDOY11], they aim "for Infer to run in 15–20 min on a diff on average".

Similarly, at Altran, an attempt to check source code into the main repository (the equivalent of `git push`) generates a requirement to prove the appropriate contracts, and the verification conditions that ensure, for example, no numeric overflow. An important requirement here is that this verification be "reasonably fast". [BS12] describes their work here as "this changes the qualitative time band for a large scale industrial project from 'Nightly' to 'Coffee'." Both Facebook and Altran argue that the primary purpose of this time requirement is to avoid 'context switch' in the developer's brain.

Further changes could be initiated by academics. "Two-hundred-terabyte maths proof is largest ever" reported Nature in May 2016[5] and wrote: "Three computer scientists have announced the largest-ever mathematics proof: a file that comes in at a whopping 200 terabytes, roughly equivalent to all the digitized text held by the US Library of Congress. The researchers have created a 68-gigabyte compressed version of their solution – which would allow anyone with about 30,000 hours of spare processor time to download, reconstruct and verify it – but a human could never hope to read through it." The results that triggered this media interest concerns the Pythagorean Triples Problem. "We consider all partitions of the set $\{1, 2, \dots\}$ of natural numbers into finitely many parts, and the question is whether always at least one part contains a Pythagorean triple (a, b, c) with $a^2+b^2 = c^2$. For example when splitting into odd and even numbers, then the odd part does not contain a Pythagorean triple (due to odd plus odd = even), but the even part contains for example $6^2 + 8^2 = 10^2$. We show that the answer is yes when partitioning into two parts, and we conjecture the answer to be yes for any finite size of the partition." [HK17] Such results triggering media interest could possibly change the situation. Another approach could be to organise, say, verification competitions at a student level. They would need to provide a stimulating social environment by being accessible to all students, and could be supported by elements such as cool prizes and free pizza.

2.4 Students' Mathematical Background

The seeming need for a solid mathematical background is often an argument against teaching formal methods. However, reflecting on the three elements of a formal method, grasping the syntax of a formal method is not more involved than understanding the syntax of a programming language: both are given by grammars. Grammars for formal methods are usually smaller than those for programming languages. However, students learn programming languages by trial and error on a computer, where the compiler/interpreter provides feedback on syntax errors. As discussed in Sect. 2.1, standard programming courses mostly

[5] Nature, 26 May 2016.

take an example-driven approach to syntax. In contrast, in formal methods students are often presented with a grammar for the syntax. For students, this often provides the first mathematical hurdle[6]. The challenge in formal methods teaching therefore lies in adopting a more example-driven style when it comes to syntax.

The semantics of a formal method is inherently mathematical in nature: in logic it is given in terms of the satisfaction of a formula by a model, process algebra utilizes structural operational semantics or denotational semantics, etc.

However, in a basic course focused upon the application of formal methods, it would be enough to point out that such formal semantics exists and to hint at its nature. The teaching challenge lies in providing an explorative approach to semantics via tools. In logic, this could follow ideas such as Tarski's world. In process algebra, one can explore processes by simulating them. In such a set-up, students could develop their own formal models and explore them, i.e., tools provide students with a similar feedback like running a computer program. Another idea would be to use a semantics compatible with the programming languages students are using. For instance in axiom-based testing, the 'axioms' can be interpreted as code in the programming language, thus utilising the students' programming background.

In an advanced course, in addition to such an explorative approach, the formal semantics itself needs to be presented. This will require a good mathematical background from the students.

Finally, the method aspect of a formal method is best presented through the use of a tool that automates the analysis in which one is interested. Running a tool would not require any mathematical background at all. Understanding the result of a method applied to a concrete example is usually immediate. An advanced course would address the mathematical details of why a method is sound.

These considerations refute the common prejudice that teaching formal methods requires students to have a profound mathematical background. An explorative teaching approach can make formal methods accessible even to students who like to program the 'artistic way'. This is supported by experience reports such as: "Engineers from entry level to principal have been able to learn TLA+ from scratch and get useful results in two to three weeks" [NRZ+15].

2.5 Personal Statements

In the order in which they were contributed, we present a number of personal statements by the co-authors.

SK. One challenge in teaching formal methods is to spark an initial interest. This is the case, because links are weak between formal methods and the current hot topics in computer science. Many students steer towards what currently is

[6] This is not eased by the often poor error messages provided by formal method tools.

perceived to dominate the future: data science and artificial intelligence, to name a just a few.

To overcome this, the formal methods community should strive to demonstrate its relevance, beyond 'classical' topics such as railway engineering. Correctness is as relevant in the new, upcoming areas of computer science as it is in the classical ones.

PK. A similar thought adding to SK: many students do not even have a clear idea of what formal methods are! They have heard of other areas such as machine learning, databases, operating systems, computer networks, compiler construction, and have an idea what is going on there. It's hard to encounter many aspects of formal methods in daily programming life, especially for a student with a limited view. So, why exactly would they pick a 'no-name' course such as "formal methods" or "model checking" over the other choices?

CD. The name of a course makes a big difference: students tend to avoid courses that already sound complicated (i.e. anything math or formal) in contrast to courses that sound 'useful' or 'applicable' or even just trendy. As a student, I had a course named "Modelling Computer Systems" that was on discrete mathematics. If it had been called "Discrete Mathematics", I'm sure it would have put several students on edge to begin with. Courses with names that contain tech buzzwords may also sound more appealing to students, such as cyber security, software testing, machine learning, artificial intelligence etc. We should consider these trendy subjects and adjust formal methods to be just as appealing, even if it means slightly adjusting course names.

MF. The lack of reliable tools that are suitable for teaching formal methods, as well as are scalable enough to demonstrate interesting and realistic use cases, creates a barrier for students. Throughout our course, we used a number of freely available formal methods and students struggled to understand the error messages and other feedback from the tools [FW20]. This kind of ambiguous feedback causes the students to lose interest and prevents them from engaging with the tools in a positive, constructive way. Furthermore, this usability issue also hinders the uptake of these tools in industry. This is somewhat of a vicious circle. Admittedly, most formal method tools are academic in nature and thus often are aimed at being good for publication. Better error messages and the like are often not prioritized that way. This causes the industrial uptake to miss, which decreases the focus again.

2.6 A Student's Personal Statement

RZ. My first introduction to formal methods was during my second year (right after introductory programming courses but before software engineering) in the GPU computing course. We used Petri nets for modelling the classic dining philosophers' problem. One of the motivations for using them was to avoid software failures. By providing a mathematical proof with Petri nets, so the professor claimed, we would be on course for success. At that time formal methods looked to me like an advanced technique in software development and a usual practice. My illusions were shattered later when another professor pointed out that it takes numerous assumptions for formal methods to work in the real world, and that often these assumptions do not apply.

3 Teaching Formal Methods — the Fun Way

In this section we collect a number of personal views and ideas on how teaching formal methods can be done the fun way. While some authors, see, e.g. [CRS+15], have written systematic accounts of the topic, here we present a number of personal statements in the order in which they were contributed.

MF. Games can be useful when it comes to teaching formal methods in the initial stages. However, to adequately demonstrate the importance of formal methods there must also be an emphasis on building and verifying software and not just on solving a puzzle, as entertaining as that may be. Of course, computer science students will find enjoyment in building systems, otherwise they would not be studying the subject. So, perhaps setting them the task of developing and verifying a simple, but realistic, model of a system would also be beneficial while encouraging them to have fun with formal methods. In this setting, games would ideally be placed at the beginning of the course as a light-weight and fun introduction.

JD. It is often difficult to motivate formal methods. Most students will not go into the construction of safety-critical systems, important though they are. Also, the specialist safety-critical companies tend to do their own training (though they would really like to have to do less!). It is perhaps easier to motivate formal methods with more common examples. The Chromium Project[7] is one example of 'mainstream' software, viz. browsers, and shows that the Chromium team is moving 'more formal'.

SK. Usually, what makes any course interesting is the applications and the transfer of knowledge from classroom to reality. However, most formal method courses rely on examples that, while interesting, are far away from what students can experience and experiment with. We often rely on examples from industry and spend quite a lot of time explaining what a particular model is supposed to achieve exactly. I feel this often distracts students. Rather than focusing on what formal methods have to offer, we get lost in technical details. This is not the case with games, especially if considering well-known ones. Usually, the rules are known and (mostly....) agreed upon already and we can focus on how a formal method can help us to get them right in our application.

Again, I strongly believe we should get away from the purely theoretical approach to teaching formal methods to beginners. At least for me, the theoretical

[7] https://www.chromium.org/Home/chromium-security/memory-safety.

advances in formal methods have always been a means to an end. In order to appreciate them, one has to experience what it means to try and reach the same end without them. This however falls short in programming education in general. Students proceed from smallish group projects to other smallish group projects, while only seldom have to experience larger refactoring, legacy code, etc. In an environment like this, formal methods are less useful. Let's teach our students what programming is like in reality: 90% of the work is reworking legacy code, fixing bugs and trying to understand why things are or are not working – by accident, this is where formal approaches could shine as well. Another aspect that could make a formal methods course interesting is to involve students in formal methods research rather than formal methods application. We used to teach formal methods by discussion software issues first and then having students try to find automatic ways to detect them, leading from simple static analysis ideas to model checking. The course has been thoroughly documented, also showing that the approach was highly motivating for students students [KKS19].

Notably, students (at least on the masters level) are able and willing to do 'actual research' in an inquiry-based course, eventually leading to publications [POKG19]. The inquiry- or research-based approach has taught students the internals of model checkers and how they can be efficiently implemented for prototypical languages.

PK. Shriram Krishnamurthi had a great Keynote at FM'19[8]. One of the main points to take away from that is that tools are a large issue. If you hit students with a full-blown industrial tool, they get frustrating error messages, because they have no idea what is going wrong (as the tool is able to understand a larger part of, e.g., a specification language than the student and raises errors related to other concepts). While it is nice to see that such tools are used in practice, they might be the wrong means to learn formal methods.

In Düsseldorf, our group has worked on an approach based on Jupyter notebooks [GL20]. It allows evaluation of smaller expressions or predicates without a state-based approach, so students can learn and experiment with the logical foundations of the language[9] where it is used to solve some logic puzzles). It can also be used to interact with B machines, so errors in a specification can be explained and documented in a nicer way (that you can replay). We think that might resolve some of the issues in teaching (but probably not all).

CD. Games are important, maybe even essential in teaching formal methods and making it fun. As a teacher of all ages from 8 years old to university level, I have found games to be one of the best tools to use when teaching. Students understand games and want to win them, naturally. When you explain to students that there is a method in which they are either guaranteed to win, or indeed a

[8] https://www.youtube.com/watch?v=UCwyOSHRBi0.
[9] https://gitlab.cs.uni-duesseldorf.de/general/stups/prob2-jupyter-kernel/-/blob/master/notebooks/tutorials/prob_solver_intro.ipynb.

method in which the second player cannot win, their interest levels peak! Students rush home to play the games against their parents and show off their new found ability.

Games can also be taught to most age groups. As said in our other paper in this volume "Appealing to their existing understanding of how the world works, using puzzles as a medium, students can quickly become comfortable using mathematical concepts such as labelled transition systems" [MOPD20].

We have had success in asking 11 year olds to draw labelled transition systems. If we start teaching them sooner, this could act as a base from which we can build upon to further their understanding later on.

MF. In our experience [FW20] the students found it difficult to bridge the gap between the theory that was taught during the course (e.g. natural deduction proofs) and the associated tool support used during the lab sessions (e.g. Coq). As a result, I am inclined to agree with SK above in that the students need to see how these methods can work in reality rather than focus too much (although it is important and should be covered at some level) on the theory.

AC. The use of tools provides a great potential for introducing fun in teaching formal methods. This is particularly true for simulation and model-checking tools, whose emphasis is in giving "life" to formal specifications rather than getting involved in the complexity of a formal proof, as it happens, instead, for theorem-proving tools. Moreover, formal methods can be applied to a large range of problems, basically any problem, well beyond the domain of computer science. These give chances to teachers to propose fun problems, such as classical mathematical puzzles as well as popular games and even video games, and to learners to select problems that are close to their personal and professional interests [CL20]. One effective approach consists of providing learners with examples of formal methods descriptions of video games and inviting them to create formal models of their favourite video games. More in general, learners may be invited to define any problem they wish, formally specify/model it and carry out analysis with the support of tools. It is actually important to blur the distinction between learner and instructor by letting the learners drive the choice of exercises and use their creativity to identify and specify potential problems and invent new games. Blurring such a distinction will also contribute to instill in students a level of self-confidence that can lead students to carry out "actual research" [POKG19] and to actively contribute to curriculum development [Zhu20].

We can conclude our discussion on the teacher's view about fun by saying that if motivation is the dimension that allows learners to build up interest in formal methods, fun is actually the essential dimension to keep learners continuously engaged, thus assuring the retention and possibly increase of their interest over time [CL20, Cer16, RCS+21]. However, it is important that the fun occurs from the perspective of the student, not the teacher, and, if it is associated with some form of competition, this much effectively fosters motivation and does not cause frustration. In fact, nothing could be worse than "fun degenerating into frustration", which could be the case when a game that is fun for the teacher is actually too complex or uninteresting for the students or, especially in the case of school children, if the outcome of competition is interpreted by the student as a form of assessment [Cer20].

NS. Fun, puzzles, games and entertainment in teaching are not the unique ingredients needed to improve formal methods education (more general – computer science and software engineering education). All these (and something else) are just ways to engage (undergraduate) students with the learning, studying, comprehension and mastering of formal methods using curiosity and amusement. We believe that the experience of individual educators and expertise of research groups in the field of formal methods popularization deserves a positive attitude from the computer science, software engineering and (even) mathematics academic community and industry.

Another opportunity (just as an example) is a competitive spirit that is so appropriate for young people (in particular – for students of computer science and software engineering departments). International competitions between formal methods tools (e.g. automated theorem provers and satisfiability solvers) are popular, useful and valuable from the industrial and research perspectives, but not from the undergraduate education perspective. Unfortunately, competitions especially designed for (undergraduate) students (like Collegiate Programming Contest[10]) are still not involved in the education process in general and in formal methods education in particular. We hope that competitions of this kind may be used better for engaging students with theory of computer science and formal methods in software engineering [SY02].

PO. I also disagree with the 'puzzle'/'games'/'card tricks' approach. I do not think that they show the usefulness and relevance of formal methods. I also use small games (lots of them!) in my second-year course, up to blackjack, but only as small "toy examples" to get to know the modeling language and tool. On the other hand, real industrial applications, as others write here, are too large and complex to include in beginner's formal methods courses. A good compromise that I use (and describe in my FMfun'19 paper) are seminal systems/algorithms that are the cornerstones of different other domains and, equally important, of today's large software systems. For example, 2-phase-commit (while simple) and Paxos (less so) are still key building blocks in today's distributed systems. I include key designs from other courses and beyond, like cryptographic protocols (modeling and breaking NSKP), distributed transactions (2PC), distributed mutual exclusion, distributed leader election, transport protocols like TCP, ABP, sliding window, and so on. This shows the relevance of formal methods on many kinds of systems, and are small enough to easily model and analyze using formal methods, but might still give students (and other professors!) an idea of the usefulness of formal methods.

One final problem with games/tricks: even if you learn how to apply your formal method to model and analyze such games, can you then apply your formal method to a real distributed system such as Paxos or a cryptographic protocol?

[10] https://icpc.baylor.edu/.

I refer to my paper “Teaching Formal Methods for Fun Using Maude” [Ölv20] in this volume for a lengthier exposition of how I think formal methods should be taught at the undergraduate level.

3.1 Summarizing the Ideas

It is obviously impossible to establish general criteria to make formal methods teaching a fun activity. Fun cannot be characterised in an objective way and can only naturally emerge from the interaction between teachers and students. In fact, the emergence of fun is affected by the personalities of individual teachers and students as well as by the interaction context in which such different personalities meet in the classroom collaborative environment. Here, different criteria have been suggested and discussed, including:

- games and puzzles may represent a light-weight and fun introduction to formal methods;
- there should be an emphasis on building and verifying software for simple, but realistic, systems;
- teaching should focus on demonstrating that tools work rather than on delivering too much theory;
- students are likely to enjoy undertaking actual research activities;
- students should be involved in curricula development.

There is a general view among the co-authors that games and puzzles can be useful when it comes to teaching formal methods in the initial stages and represent a light-weight and fun introduction (MF, CD, AC). It is important to note that this view includes former formal methods students who became formal methods teachers [MOPD20]. Games may be also associated with some form of competition (AC, NS), which may be within-class (AC) or in terms of participation at an international context (NS). Games and puzzles are also a great tool to start formal methods education early, even by teaching to school level children, as young as 10–11 (CD, AC). Competition can also be beneficial in the context of school children, but should to carefully planned in order to avoid being interpreted by the student as a form of assessment, which therefore inhibits rather than motivates the students [Cer20].

In addition, there must also be some emphasis on building and verifying software (MF). However, such a connection with reality should be established in the right form to keep in line with the fun determined by the game-based approach. In fact, giving students the task of developing and verifying a simple, but realistic, model of a system would be beneficial while encouraging them to have fun with formal methods (MF). However, on the one hand, realistic, industrial systems are often far away from what students can experience and experiment with (SK) and most students will not go into the construction of safety-critical systems, important though they are (JD). On the other hand, the specialist safety-critical companies tend to do their own training (JD), which may provide a very different perspective from what students learn in formal methods courses. Moreover, focusing on examples from industry is very time consuming and often involves heavy technical details and, as a consequence, may be distractive rather than beneficial (SK). Instead, it might be more effective to motivate formal methods with more common, but still realistic examples, such as the Chromium Project (JD).

There is a general agreement among the co-authors that students need to see how formal methods work in reality using tools rather than focusing too much on the theory (SK, MF, PK, AC, PO). However, making students use industrial tools may result in heavy frustration. While it is nice to see that such tools are used in practice, they might be the wrong means to learn formal methods (PK).

An final aspect that could make a formal methods course interesting is to involve students in formal methods research rather than formal methods application (SK). In fact, students' publication are often highly appreciated [POKG19, Zhu20].

4 Increasing Visibility of Formal Methods Throughout the Curriculum

In common computer science and software engineering curricula, formal methods play a minor role. There are at most one or two specialized courses focusing on teaching formal methods. Often, these courses are only weakly linked to the rest of the curriculum.

Formal methods fail to link to the current hot topics in computer science and software engineering, both in teaching and research. In consequence, even students with considerable interest in software engineering are drawn towards courses such as data science, machine learning or artificial intelligence. However, now that artificial intelligence and machine learning techniques find their way into safety critical systems (such as autonomous cars), correctness considerations become more important every day.

The 'winner-takes-all' nature of today's software industry (where essentially *one* product/service in each category 'wins' and makes billions, and other solutions fade away, e.g., Facebook for social media; Google for search engines, eBay for online auctions, Zoom for online discussions/teaching/meetings) justifies an upfront investment in system quality. We note that major firms like Google [SvGJ+15], Facebook [DFLO19], and Amazon [NRZ+15] are all doing this, but this has yet to feed through to their hiring practices, or to students' perceptions of what they need to get a job at these favoured employers.

In consequence, an ideal integration of formal methods into a computer science or software engineering curriculum should first and foremost strive to present formal methods as a quality assurance tool to be used in other areas, be it embedded systems engineering or machine learning. This first contact to formal methods would aim at teaching usage scenarios as well as techniques and how they are to be deployed.

We believe that showing the benefit of formal methods by discussing applications to other areas will achieve two goals. First, it ensures code quality and system functionality are considered as critical. Furthermore, this initial contact to formal methods might spark an interest into their development and improvement. Both topics could then be a part of dedicated courses in formal methods.

While such a 'casual' approach would be ideal, it would require colleagues to be willing and to be able to teach small units on formal methods. This might be an unrealistic assumption. Organising 'guest sessions' from formal methods experts might be a way forward.

To gain an acceptance of having more formal methods visibility in a university curriculum, we need to persuade first and foremost our colleagues. Ultimately they decide whether/how/how much formal methods a university curriculum could/must contain. There is huge competition for places on a curriculum between the different specialties/fields. At least the older colleagues may remember times when formal methods were not too useful.

The 2013 "Curriculum Guidelines for Undergraduate Degree Programs in Computer Science" [ACM13] lists 18 "Knowledge Areas". In the following, we make a number of suggestions for formal method units in some of these areas:

AL-Algorithms and Complexity: formal verification of algorithms; model checking algorithms.

DS-Discrete Structures: logic, modelling, semantic foundations of formal methods.

HCI-Human-Computer Interaction: mode confusion problems; formal analysis of user dialogs; cognitive models.

IAS-Information Assurance and Security: formal analysis of security protocols.

IM-Information Management: specifying and analyzing both the correctness and the performance of cloud storage systems.

NC-Networking and Communication: protocol verification.

OS-Operating Systems: parallel modelling; scheduling.

PBD-Platform-based Development: formal model based development.

PD-Parallel and Distributed Computing: process calculi; Petri nets.

PL-Programming Languages: how to analyse software written in a specific programming paradigm; compiler correctness; semantics of programming languages; program correctness.

5 Syllabus of a Compulsory Formal Methods Course

Besides increasing the visibility of formal methods throughout all courses and also having specialised advanced courses on formal methods, we suggest that curricula for computer science and software engineering should include a compulsory formal methods course.

The target audience for such a compulsory formal methods course would be the complete cohort of computer science/software engineering students in year 2 or year 3 of a 3-year BSc degree programme.

Due to the wealth of available formal methods, we refrain from proposing a unified or 'standard' syllabus. Local expertise in specific formal methods and application domains should be taken into account. Therefore, we rather capture the essence of an ideal course in a generic way:

Introduction.

- The role of formal methods in the context of software engineering, see, e.g., Roggenbach et al. [RCS+21], Chapter 1, for a thorough discussion, and Barnes [Bar11] for a comparative case study.
- Success stories of formal methods, see, e.g., Roggenbach et al. [RCS+21] for a compilation of such stories, another good source is Section 1.3.4 of Garavel's report [GG13];
- Relating formal methods to current trends in computer science, such as machine learning, where one can use machine learning to improve formal methods [ALB18], or, a nascent field but one that is growing in importance and has already attracted the attention of ISO in the draft TR 24029-2, the application of formal methods to big data [vdA16,Cam14,MLM18] or to machine learning [HKW17,SKS19,WPW+18].

Main Part. The main part should offer one or two formal methods of different nature, e.g. a "model-oriented" and a "property-oriented" one, cf. [Win90] for further discussion of this classification; in order to demonstrate the 'universality' of formal methods, it would appear useful to draw examples from different domains.

The following topics (listed in no particular order) should be covered:

- Modelling: going from the informal to the formal; traceability; validation of models.
- Language design: explaining how the language of a formal method is designed for specific purposes (what are essentials necessary for expressivity, what is syntactic sugar easing the life of the specifier?).
- Semantics: presenting just the essentials – this needs to be one topic among many rather than the dominating one, as happens too often in current practice.
- Software engineering context: demonstrating that formal methods are applicable throughout the whole software lifecycle, e.g., in analysing designs, in software verification, testing from formal models.

- Method: systematically using tools to illustrate the 'method' aspect.
- Application domains: illustrate the reach of formal methods by selecting examples from different application domains. Safety, security, human-computer interaction, e-contracts, and non-computer areas (biological systems, ecology, chemistry) are some possible examples.
  Traditionally, formal methods teaching advocates the use of formal methods for safety-critical systems. Formal methods are of course super-important for those systems, but experience in class (and otherwise) suggests that this does not inspire and is almost counterproductive: most students do not foresee themselves designing the quite narrow range of safety-critical systems we tend to use as example (airplanes, cars, medical devices, etc.); focusing almost exclusively on safety-critical systems can actually be counterproductive as it (can be perceived to) send signals that formal methods are only usable for such systems.
  As cybersecurity failures are much in the news, we might look at these and see how formal methods might have found these (e.g. Heartbleed), or are being used (e.g. Chromium), as a way of emphasising the mainstream utility of formal methods.

Conclusion – Reflection on Formal Methods. We present below some items of reflective nature that ought to be addressed at the end of a formal methods course.

- General limitations: what formal methods can offer, what formal methods cannot deliver, e.g., based on Levenson's provocative article "Are You Sure Your Software Will Not Kill Anyone?" [Lev20].
- Scalability: why formal methods work on toy examples but their application might become impossible for technical reasons when it comes to real life challenges, see, e.g., [RMS+12] and [JMN+14]. [RMS+12] shows a formal methods in its early stages, where it can barely verify a toy example; [JMN+14] shows how, after two further years of research, with the help of abstractions it is possible to verify a real world example with the very same approach.
- Costs/benefits: what the cost and financial benefits of formal methods are [Bar11]. The key insight "Formal methods are surprisingly feasible for mainstream software development and give good return on investment." from Newcombe et al. [NRZ+15] and Amazon's "We can now use automated reasoning to provide inexpensive and provable assurance to customers" from J. Backes et al. [BBC+19] are probably a 'must have'!
- Acceptance: current uptake of formal methods in industry and reasons for the low acceptance.
- Current trends: where one expects the field of formal methods to be in, say, a decade.

Each lecturer will have her/his own subjective view concerning the above list of topics. Probably they offer a good point for discussion with students. The systematic element underlying them is that they ought to be addressed at the end of a formal methods course.

Learning Outcomes. Such a course would provide the learning outcomes that students

- understand the thinking behind formal methods and how it differs from ad-hoc programming;
- are fluent in the application of one or two formal methods to academic examples;
- are able to estimate the potential of formal methods with concrete challenges;
- are able to critically compare different formal approaches and choose the most appropriate for a given, specific application.

6 How to Assess Our Teaching Efforts?

Having introduced changes to teaching, it is important to assess if they have been successful. In this section, we collect a number of ideas as to how this could be done. In the order in which they were contributed, we present a number of personal statements.

MF. The obvious measurement is to compare exam results year after year, assuming that the same person teaches the course before and after any changes are made. We are working towards making some changes to our course that we could compare against the previous years' results. However, it is also important to survey the students before and after the course as well as during the lab sessions to really understand how they are progressing and how effective the notes, teaching and lab sessions are in improving their formal methods expertise.

CD. As a teacher on a Degree Apprenticeship programme, I think one such method of assessing our own teaching methods, is to actually assess the students' level of understanding by getting them to apply formal methods in their workplace: students on our programme are employed. Often, when we teach formal methods, our students have never seen them before. We tasked our students with producing a work-based portfolio where they have to apply discrete mathematics to their workplace. Whilst some students struggle with the task, for most of them it becomes apparent how beneficial it is in the workplace. Sometimes it even highlights issues with the existing systems logic. In my opinion this is the best outcome and therefore would demonstrate that we have been teaching successfully.

SK. The formal methods community ought to reflect on what it wants to achieve in teaching. Ultimately, there is no use in being able to enumerate different formal methods and just being able to use them if you don't see any reason to do so. Rather, I am in favour of indeed trying to change (and measure/evaluate) students' opinions and attitudes.

Employing a formal approach to software engineering is all about the resulting quality of the product. Thus, a formal methods course needs to change students' perceptions about software as a product that is used in different applications and situations – eventually, even in safety critical ones. Nobody would cross a bridge that seems like it might collapse. At the same time, delivering software that is known to cease working under certain conditions has become quite accepted. Once students gain an awareness and consciousness for quality aspects of software, formal methods (and the effort to use them) will appear more beneficial.

MR. In my teaching experience, students best learn those topics that they like to do, that they can try themselves, and that provide them with a feeling of achievement. For teaching practice in formal methods that means that we ought to run supporting lab classes. These would offer meaningful examples on which students can successfully apply a formal method, or explore why some specific formal method fails. In my view, lab tasks would be well-designed if, say, 80% of the students can solve them, i.e., offer them a sense of achievement.

The other objective would be to educate the majority of computer science students in such a way that they are capable of applying formal methods in their future careers in industry. This could be evaluated by looking at dissertations: do the majority of them report on the application of formal methods when the project concerns software development?

PK. One criterion could be the number of students that are interested in writing their dissertation in the field of formal methods. In particular, our experience is that while formal methods are not in high demand with students, the ones who finish our formal methods courses usually are willing to gain an expert level of knowledge. Many students stay interested, once they have developed an appetite for formal methods.

AC. Assessing the effect of teaching changes in standard formal methods courses is a tricky task for a number of reasons:

1. classes are normally small;
2. even within the small group there is often a large variety in background and interest of the students;
3. although students might be interested and even successful in using formal methods, in their future research or work goals they are driven by more trendy areas and topics, where there is little place for the use of formal methods.

Reason 1 prevents us from collecting enough data to allow us to produce statistically significant results. It is therefore more important to informally collect personal opinions from students through discussions, open-ended questionnaires and interviews, rather than analysing numerical data such as grades and percentage of successful students.

Reason 2 requires an initial assessment of the students to be compared with the final objectives that they achieve at the and of the course (see MF's statement earlier in this section). A possible form of initial assessment is a questionnaire to be administered during the very first course class. The questionnaire should aim at the assessment of

- mathematical background;
- logical and problem solving skills;
- experience with the logic and functional programming paradigms;
- knowledge of the software engineering concepts that are central in formal methods, such as specification, testing, verification, validation, assurance.
- knowledge of basic logical and set-theoretic concepts such as syntax, semantics, theorem, proof, function and more specific computability concepts such as decidability, enumerability, undecidability.
- perception of more "exotic" formal methods concepts such as system state and concurrent system.

Due to Reason 3, looking at dissertations or careers of former students does not really provide a measure of the achievement of learning objective. In fact, students' pragmatics in looking for a thesis topic or choosing their professional career may clash with their academic interests.

6.1 Summarizing the Ideas

Assessment is often an exercise of producing numbers that can be compared over several academic years. Here, different criteria have been suggested and discussed, including:

- exam results of a particular course;
- number of dissertations in which formal methods are applied; and
- number of dissertations in the area of formal methods.

AC provided arguments why one should look at such numbers with care.

For teaching a formal method it has been suggested to closely survey students during the course (MF), and to design lab classes with 'guaranteed success', i.e., which are barely contributing to a differentiation between students in form of marks (MR).

A slightly deeper looking approach would be to look at students' opinions and attitudes and see how they change over time (SK).

7 Conclusion and Outlook

In this white paper, we have analysed why formal methods are seldom prominently included in computer science and software engineering curricula. One often heard reason for this is that they fail to attract students. However, we believe that students often just have misconceptions about formal methods. Also, the 'coolness factor' of formal methods is low. Finally, formal methods are not visibly used by industry. It is a myth that formal methods teaching on a basic level would require a particularly strong mathematical background. We provided a number of ideas on how to make formal methods more attractive to students and gave examples of the uptake of formal methods in industry beyond the critical systems sector.

In the spirit of the workshop "Formal Methods – Fun for Everybody", this paper has collected a number of 'sparkling ideas' that aim at improving the situation summarised above. We grouped such ideas into four categories, namely individual teaching delivery, cf. Sect. 3, making formal methods visible throughout the syllabus, cf. Sect. 4, the proposal of a compulsory formal methods course, cf. Sect. 5, and ideas about how to measure the effect of teaching changes, cf. Sect. 6.

With this white paper a start has been made to make formal method teaching more popular. The ideas and arguments presented are ready to be picked up in order to improve existing courses, to design new courses, and to make formal methods more prominent in academic curricula. The participants of the 2019 workshop were enthusiastic about this topic, and we hope to have shared some of this enthusiasm with the reader. Let's turn this into a wider movement!

References

[ACM13] ACM. Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science (2013). http://dx.doi.org/10.1145/2534860

[ACM15] ACM. Software engineering 2014: Curriculum guidelines for undergraduate degree programs in computer science (2015). https://doi.org/10.1145/2965631

[ALB18] Amrani, M., Lucio, L., Bibal, A.: ML + FV = ♡? A survey on the application of machine learning to formal verification. arXiv Software Engineering (2018)

[Bar11] Barnes, J.E.: Experiences in the industrial use of formal methods. In: Romanovsky, A., Jones, C., Bendiposto, J., Leuschel, M., (eds.) AVoCS 2011. Electronic Communications of the EASST (2011)

[BBC+19] Backes, J., Bolignano, P., Cook, B., Gacek, A., Luckow, K.S., Rungta, N., Schaef, M., Schlesinger, C., Tanash, R., Varming, C., Whalen, M.: One-click formal methods. IEEE Softw. **36**(6), 61–65 (2019)

[BDK+06] Brakman, H., Driessen, V., Kavuma, J., Bijvank, L.N., Vermolen, S.: Supporting formal method teaching with real-life protocols. In: Formal Methods in the Teaching Lab (2006). http://www4.di.uminho.pt/FME-SoE/FMEd06/Preprints.pdf

[BLA+09] Blanco, J., Losano, L., Aguirre, N., Novaira, M.M., Permigiani, S., Scilingo, G.: An introductory course on programming based on formal specification and program calculation. SIGCSE Bull. **41**(2), 31–37 (2009)

[Bou09] Boute, R.: Teaching and practicing computer science at the university level. SIGCSE Bull. **41**(2), 24–30 (2009)

[BS12] Brain, M., Schanda, F.: A lightweight technique for distributed and incremental program verification. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 114–129. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_10

[Cam14] Camilli, M.: Formal verification problems in a big data world: towards a mighty synergy. In: Proceedings of ICSE 2014, pp. 638–641. ACM (2014)

[CC82] Cooper, D., Clancy, M.: Oh! Pascal. W.W. Norton & Company Inc., New York (1982)

[CCC+18] Chudnov, A., et al.: Continuous formal verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 430–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_26

[CDOY11] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011)

[Cer16] Cerone, A.: Human-oriented formal modelling of human-computer interaction: practitioners' and students' perspectives. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 232–241. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_17

[Cer20] Cerone, A.: From stories to concurrency: How children can play with formal methods. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 191–207. Springer, Cham (2017)

[CL20] Cerone, A., Lermer, K.R.: Adapting to different types of target audience in teaching formal methods. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 106–123. Springer, Cham (2017)

[CRS+15] Cerone, A., Roggenbach, M., Schlingloff, B.-H., Schneider, G., Shaikh, S.A.: Teaching formal methods for software engineering - ten principles (2015). https://www.informaticadidactica.de/uploads/Artikel/Schlinghoff2015/Schlinghoff2015.pdf

[DD07] Deitel, P.J., Deitel, H.M.: Java - How to Program, 7th edn. Pearson Education Inc., Upper Saddle River (2007)

[DFLO19] Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019)

[DS18] Dewar, R.B.K., Schonberg, E.: Computer science education: Where are the software engineers of tomorrow? CROSSTALK - The Journal of Defense Software Engineering (2018)

[Flo67] Floyd, R.W.: Assigning meaning to programs. Math. Aspects Comput. Sci. **19**, 19–32 (1967)

[FW20] Farrell, M., Wu, H.: When the student becomes the teacher. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 208–217. Springer, Cham (2017)

[GG13] Garavel, H., Graf, S.: Formal Methods for Safe and Secure Computers Systems. Federal Office for Information Security (2013). https://www.bsi.bund.de/DE/Publikationen/Studien/Formal_Methods_Study_875/study_875.html

[Gib08] Paul Gibson, J.: Formal methods: never too young to start. In: Proceedings of FORMED 2008, pp. 151–160 (2008)

[GJS+20] Gosling, J., et al.: The Java language specification - Java SE 14 Edition. Technical Report JSR-389 Java SE 2014, Oracle America, February 2020

[GL20] Geleßus, D., Leuschel, M.: ProB and Jupyter for logic, set theory, theoretical computer science and formal methods. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 248–254. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_19

[Gla00] Glass, R.L.: A new answer to "how important is mathematics to the software practitioner?". IEEE Softw. **17**(6), 136 (2000)

[GM13] Gnesi, S., Margaria, T.: Some Trends in Formal Methods Applications to Railway Signaling, pp. 61–84 (2013)

[HK17] Heule, M.J.H., Kullmann, O.: The science of brute force. Commun. ACM **60**(8), 70–79 (2017)

[HKW17] Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1

[HW73] Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Inf. **2**, 335–355 (1973)

[ISO90] ISO 7185:1990 Information technology - Programming languages - Pascal (1990)

[JMN+14] James, P., Moller, F., Nga, N.H., Roggenbach, M., Schneider, S.A., Treharne, H.: Techniques for modelling and verifying railway interlockings. Int. J. Softw. Tools Technol. Transf. **16**(6), 685–711 (2014)

[KKS19] Krings, S., Körner, P., Schmidt, J.: Experience report on an inquiry-based course on model checking. In: Tagungsband des 16. Workshops zu Software Engineering im Unterricht der Hochschulen, CEUR, vol. 2358 (2019)

[Lev20] Leveson, N.: Are you sure your software will not kill anyone? Commun. ACM **63**(2), 25–28 (2020)

[MC15] McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)

[MLM18] Mandrioli, C., Leva, A., Maggio, M.: Dynamic models for the formal verification of big data applications via stochastic model checking. In: Proceedings of CCTA 2018, pp. 1466–1471. IEEE Computer Society (2018)

[MOPD20] Moller, F., O'Reilly, L., Powell, S., Denner, C.: Teaching them early: formal methods in school. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 173–190. Springer, Cham (2017)

[NRZ+15] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)

[Ölv20] Ölveczky, P.: Teaching formal methods for fun using Maude. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 58–91. Springer, Cham (2017)

[Pat94] Pattis, R.E.: Teaching EBNF first in CS 1. In: Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education, SIGCSE 1994, New York, NY, USA, pp. 300–303. Association for Computing Machinery (1994)

[POKG19] Petrasch, J., Oepen, J.-H., Krings, S., Gericke, M.: Writing a model checker in 80 days: reusable libraries and custom implementation. In: Proceedings of AVoCS 2018, vol. 76, Electronic Communications of the EASST (2019)

[RCS+21] Roggenbach, M., Cerone, A., Schlingloff, B.-H., Schneider, G., Shaikh, S.A.: Formal Methods for Software Engineering. Springer, Switzerland (2021)

[RMS+12] Roggenbach, M., Moller, F., Schneider, S., Treharne, H., Nguyen, H.N.: Railway modelling in CSP||B: the double junction case study. ECEASST, 53 (2012)

[Sek06] Sekerinski, E.: Teaching the mathematics of software design. In: Formal Methods in the Teaching Lab (2006). http://www4.di.uminho.pt/FME-SoE/FMEd06/Preprints.pdf

[SKS19] Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Proceedings of HSCC 2019, pp. 147–156. ACM (2019)

[SvGJ+15] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., Winter, C.: Tricorder: building a program analysis ecosystem. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 598–608 (2015)

[SY02] Shilov, N.V., Yi, K.: Engaging students with theory through ACM collegiate programming contests. Commun. ACM **45**(9), 98–101 (2002)

[vdA16] van der Aalst, W.: Process Mining - Data Science in Action, 2nd edn. Springer, Heidelberg (2016)

[vRtPdt20] van Rossum, G., the Python development team: the Python Language Reference - Release 3.8.3. Python Software Foundation, June 2020. Retrieved 2020–06-15

[Win90] Wing, J.: A specifier's introduction to formal methods. IEEE Comput. **23**(9), 8–22 (1990)

[WPW+18] Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of Sec 2018, pp. 1599–1614. ACM (2018)

[Zhu20] Zhumagambetov, R.: Teaching formal methods in academia: a systematic literature review. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 218–226. Springer, Cham (2017)