The Role of Formal Methods in Computer Science Education

By Maurice ter Beek, *CNR–ISTI*, Manfred Broy, *Technische Universität München*, and Brijesh Dongol, *University of Surrey*

This piece points out the key position of formal methods in Computer Science (CS) education, which must thus be reflected in any CS curriculum as a knowledge area rather than as elective topics in distinct knowledge areas. This is confirmed by the increasing use of formal methods in industry [4]—not limited to safety-critical domains. First, we indicate the importance of formal methods thinking in CS education [17], since this provides the necessary rigor in reasoning about software, its specification, its verification, and its correctness—all fundamental skills for future software developers. Then, we argue that every computer scientist needs to know formal methods [6], since the skills and knowledge acquired in this way provide the indispensable solid foundation that forms the backbone of CS practice. Finally, we underline that teaching formal methods need not come at the cost of displacing other engineering aspects of CS that are already widely accepted as essential. On the contrary, formal methods have the potential to support and strengthen the presentation and knowledge in all these subdisciplines. We provide suggestions for educators on how to incorporate formal methods into CS education.

INTRODUCTION

This article summarizes discussion and evidence brought forward in three white papers on the role of formal methods in CS education, which were written by 35 authors [4,6,17]. Formal methods have multiple characterizations in the literature as languages and techniques (and tools) based on rigorous mathematical foundations for the specification, development, and (manual or automated) analysis and verification of software and hardware systems [4,6,12,17,24,41].

Let us start by explaining what a formal method is. According to IEEE, "software engineering methods provide an organized, systematic approach for specifying, designing, constructing, testing, and verifying the resulting software products and associated work items involved in developing computer software applications. The methods impose a certain structure, set of steps, practices, and procedures on the software engineering effort to make it more methodical, repeatable, and more success-oriented" [27]. A method is called "formal" if its set of steps is applied using formal techniques such that the correctness of the result of the application is formally justified. Let us look at a very basic example. Using Hoare-style program annotation, we write the formal proposition

$$\{Q\}\ S\ \{R\}$$

to express that from any state in which the assertion Q (i.e., the precondition) holds, execution of the program S (if it terminates) leads to a state in which the assertion R (i.e., the postcondition) holds. There is a set of proof rules to give a formal proof for this proposition. In this example of a formal method, the effect of the program is formalized by formalizing the assertions Q and R and the proposition {Q} S {R} is a formal statement that is formally verified. Nevertheless, this idea can also be applied as a semi-formal method, by formalizing the assertions Q and R and then giving informal arguments to argue that {Q} S {R} holds. The method can also be applied fully informally by giving informal descriptions of Q and R and giving informal arguments to show that these descriptions are fulfilled. Having seen and understood the formal method, both the semi-formal and the informal method are not only clearly understood—they can be applied with much more care.

```
1  function factorial(n: int): int // Factorial specification
2  requires n >= 0
3  { if 0 <= n <= 1  then 1 else n * factorial(n-1) }
4
5
6  method Factorial(x: int) returns (r: int)
7  requires x >= 0 // Precondition
8  ensures r == factorial(x) // Postcondition
9  {
10     var k := x;
11     r := 1;
12     while (k != 0)
13     invariant r * factorial(k) == factorial(x) // Loop invariant
14     {
15       r := r * k;
16       k := k - 1;
17     }
18 }
```

Figure 1: A proof of correctness of a program to compute the factorial in Dafny.

Many tools that support (semi-)automated verification have been developed. Provers such as Dafny [31], which enable verification of Hoare-style annotations, are routinely used in industry [11]. In Figure 1, we give the example of a Dafny method Factorial for computing the factorial of the given input. The precondition (denoted by the requires keyword) assumes that the input value x is at least 0, while the postcondition (denoted by the ensures keyword) guarantees that the return value r is indeed the factorial of the input x, where the factorial is specified by the recursive function definition (factorial) in lines 1–3. This proof additionally requires the support of a loop invariant in line 13, which must be supplied by the programmer. All other aspects of the proof, e.g., the introduction of intermediate assertions, are automatically handled by Dafny.

Ongoing improvements in such formal verification tools means that they are often used to teach formal methods in courses around the world [21], since they allow students to quickly link theory with practice. From an educator's perspective, the ability to explain concepts such as pre/postconditions and invariants using a tool that can check their correctness is compelling proposition, since this immediately improves a student's engagement with these seemingly abstract concepts.

Formal methods come in many shapes and sizes, ranging from lightweight static analysis to heavyweight interactive theorem proving (see Figure 2). We emphasize the role of formal methods in complementing existing validation and verification techniques like testing and simulation. This may include the use of partiality [28] (of specifications, languages, modeling, and analysis) to reduce the cost of deployment and increase tractability. The distinguishing feature of formal methods is their capability to enable the systematic usage of formal foundations in CS in engineering tasks such as guaranteeing the absence of misinterpretations of requirements in a system under development.

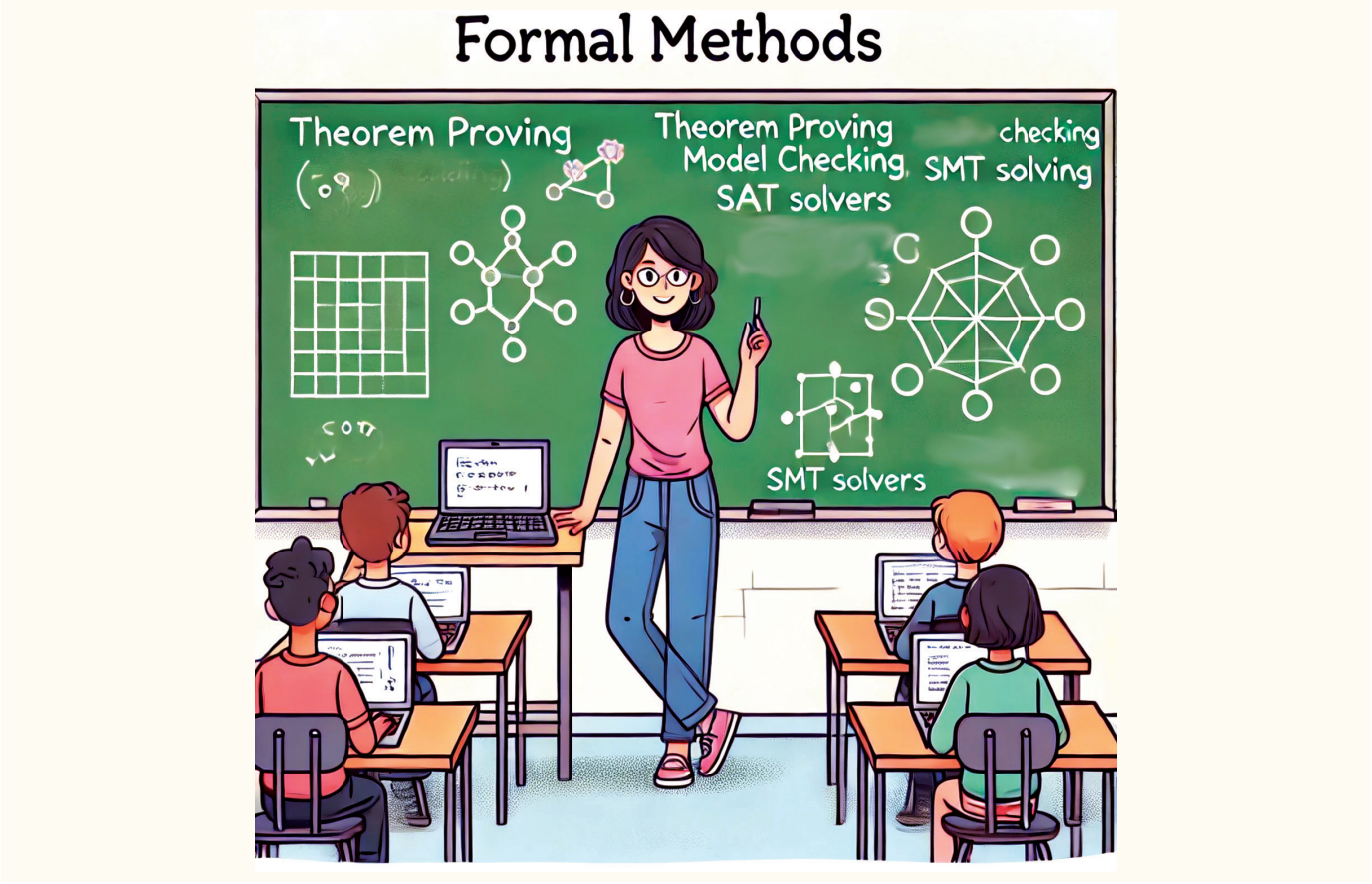

Figure 2: Formal methods teaching (Image courtesy of Luigia Petre, generated using Dall-E).

Obviously, there are various positions of the experts in our field on the significance of formal methods. In the following, we formulate a plea for introducing formal methods as a core subject into CS curricula. We provide supporting evidence from renowned experts in the field. Currently, formal methods do not appear in the ACM CS2023 curriculum to the extent that reflects their fundamental role in CS and the benefits that a targeted education in formal methods brings. In particular, formal methods do not appear as a separate knowledge area but only as elective topics in some knowledge units in two distinct knowledge areas (namely Foundations of Programming Languages and Software Engineering). The abovementioned white papers make the following four key points.

1. Formal methods are applicable in numerous industrial domains, not limited to safety-critical applications.
2. Formal methods cover key tasks in CS such as requirements engineering, specification, algorithmic problem solving, verification, model-driven engineering, security, and many more.
3. Every CS graduate needs to have a fundamental education in formal methods.
4. The current offering of formal methods in the CS curricula and, in particular, in CS2023 is inadequate.

ACM CS2023 would have been the ideal time and place to adjust the way we teach CS. There are mature tools and proofs of concept available and the possibility of designing coherent teaching paths.

Computer Science, namely the science of solving problems by software and software-intensive systems, provides the knowledge and skills to understand and capture precisely what a situation requires, and then develops a program providing a formal solution. Two of the most fundamental skills of a computer scientist, those of abstraction and formalization, are effectively addressed by formal methods. They provide the rigor for reasoning in precise terms about goals, such as specification, validation, and verification, thus guaranteeing adequacy, accuracy, and correctness of designs and implementations.

Formal methods are becoming more widely applied in industry, from eliciting requirements and early design to deployment, configuration, and runtime monitoring. Evidence of successfully applying formal methods in industry ranges from stories in the safety-critical domain, such as railways and other transportation domains, to areas such as lithography manufacturing and cloud security in e-commerce, for example. "Papers and testimonies come from representatives who, either directly or indirectly, use or have used formal methods in their industrial project endeavors. Importantly, they are spread geographically, including Europe, Asia, North and South America, and involve well-known worldwide companies such as Alstom, Amazon, ASML, Bang & Olufsen, Boeing, Collins Aerospace, Embraer, Facebook, Google, Huawei, IBM, Intel, Microsoft, Motorola, Oracle, Siemens, and Volvo" [4].

Therefore, at the very least formal methods thinking, enabling the application of formal methods in lightweight, practical, and accessible ways, should be part of the recommended curriculum for every CS student. We have advocated for formal methods thinking at three levels of expertise: informal (Level 1), semi-formal (Level 2) and fully formal (Level 3) [17]. Students who train only in this 'thinking' will become much better programmers with a deeper understanding of their tasks, since it involves annotating programs with assertions that describe what is true at particular points of a program. Our basic example of the formal method of verifying {Q} S {R} demonstrates clearly that knowing the formal method supports the semi-formal or informal use of the idea. Formal methods thinking (in particular, Levels 1 and 2) can be incorporated into any CS curriculum without introducing additional teaching hours. Moreover, there are students who, exposed to those ideas, will be ideally positioned to study more: why the techniques work; how they can be automated; and how new ones can be created. Thus, teaching Levels 1 and 2 supports subsequent (optional) courses that include topics such as formal semantics and proof-automation techniques (Level 3).

Formal methods can assist in teaching programming to novices more effectively than by informal reasoning and testing. Formal methods explain design patterns, model-driven engineering, software architecture, software product lines, requirements engineering, and security, being complementary to these CS fields. Formalisms concisely and precisely express underlying fundamental design principles and equip programmers with a tool to handle related problems.

ACM CS2023 would have been the ideal time and place to adjust the way we teach CS. There are mature tools and proofs of concept available and the possibility of designing coherent teaching paths. Importantly, this can be done without displacing the other 'engineering' aspects of CS already widely accepted as essential. Support for teachers is available, for instance via Formal Methods Europe (FME) [20,22].

The next four sections each deal with a different one of the aforementioned four key points, the first three of which are based on the above mentioned three white papers. First, we emphasize the importance of formal methods thinking in CS education. Then, we underline the importance of knowing formal methods for CS graduates, as witnessed and testified by practitioners using formal methods in industry in the subsequent section, after which we aim to convince the reader that teaching formal methods need not come at the cost of displacing other engineering aspects of CS. Finally, we conclude by wrapping up our position on the role of formal methods in CS education. All quotes reported hereafter stem from the three white papers, unless indicated otherwise.

FORMAL METHODS THINKING

Formal methods thinking offers distinct advantages over relying solely on an intuitive understanding of program execution. Testing, for instance, has limitations: it may not guarantee correctness, subjective judgment is needed for some outputs, and untried cases can hide errors, at least as long as validated formal specifications are not worked out. Courses that teach formal methods provide students with an independent understanding of programs by proving their satisfaction of specifications for all inputs, complementing testing capabilities. While testing remains crucial, formal methods augment it by reducing the likelihood of mistakes, improving specification precision, and fostering reflective design practices, thus instilling in students "a mindset of reflecting on our designs and checking (or verifying) that the intentions (or requirements) are met" [17]. Formal methods enhance one's argumentation skills and lay the foundation for solid reasoning about systems behavior.

Formal-methods thinking is especially beneficial in programming, enhancing students' ability to model computational problems and comprehend their code, thereby improving their programming skills. However, formal methods thinking extends beyond programming and can be applied throughout the software life cycle phases.

Most CS curricula initially cover mathematical foundations alongside introductions to programming, algorithms, and data structures. As students progress, they specialize in various applications and CS knowledge areas like databases, security, concurrency, networks, Artificial Intelligence (AI), etc. The level of foundational mathematics varies based on later specializations, encompassing topics such as discrete mathematics, logic, probability theory, control theory, and linear algebra. While the discussion of the appropriate level of mathematics for the average programmer is not the focus here, a reasonable argument could be made that people should not be writing programs whose functionality requires discrete mathematics to describe it, unless they have some command of discrete mathematics themselves. We believe that formal methods thinking should extend beyond discrete mathematics to address also the complex dynamics and behavior of modern systems.

Formal-methods thinking is especially beneficial in programming, enhancing students' ability to model computational problems and comprehend their code, thereby improving their programming skills. However, formal methods thinking extends beyond programming and can be applied throughout the software life cycle phases:

- **Analysis:** system modeling and requirements elicitation and specification
- **Product Design:** specification of functional and quality requirements
- **Validation:** analyzing requirements on validity and comprehensiveness
- **Software Development:** writing clean code and documentation, establishing correctness;
- **Verification:** specification-based testing, complementing testing with formal reasoning
- **Maintenance:** system adaptation, error correction, quality improvement

Formal methods are increasingly employed in soft-ware engineering, cybersecurity, and computer networking—to name just a few. The central challenge addressed by formal methods is "the need for precision and rigor in modelling and analyzing computer systems and software" [17].

KNOWING FORMAL METHODS

Early computing pioneers like Turing and von Neumann recognized the importance of reasoning about programs, publishing papers in the 1940s that demonstrated the possibility of recording proofs for program correctness. Over the decades, researchers have enhanced the tractability of specification and reasoning, developing supporting software tools, many of which still require users to understand the underlying formalisms.

Just like structural engineers do not always apply their most rigorous techniques, not all software needs to be developed formally. But just like structural engineers need to learn the foundational methods supporting documentation and reasoning about their designs, "software engineers must learn how precise specifications are constructed and how their key design decisions are subject to rigorous justification" [6]. No structural engineer would be permitted to work on the design of a bridge without a solid understanding of the relevant mathematical theories.

Building reliable systems necessitates rigorous development approaches rooted in abstract comprehensive models, unambiguous specifications, thorough testing, and verification methods to ensure system requirements are met. While software systems do not endure wear and tear, changing environments introduce new requirements, necessitating long-term maintenance plans. CS is an independent engineering discipline, which draws from and interacts with other engineering disciplines, but relies heavily on formal techniques. Computer programs interface with the physical world and impact the real world, from controlling machinery to managing energy distribution. In many such domains, formal descriptions are indispensable to predict and prevent unintended consequences with significant responsibility. Creating these formal descriptions requires skills in abstraction, rigor, and a clear understanding of the models' semantics. It seems irresponsible to let anyone design high-impact computer control

programs without suitable training in formal methods to mitigate potential risks, regardless of whether these are applied to systems from the safety-critical domain.

Fortunately, programmers commonly use some (lightweight) formal methods (thinking) in their daily work, such as type systems for defining formal requirements on value expressions and checking compliance of the values produced by expressions with given types. This also helps with program decomposition and structuring. Some programming languages have type systems that demand a formal understanding of types and the type-checking process, and the ability to apply abstraction. In practice, often formal techniques are no longer called 'formal methods' since they are so smoothly integrated into the engineering. Formal methods can act as a bridge between pure mathematical foundations and general software development. "Formal methods thinking consists of describing a system to be understood or designed in terms of fundamental discrete mathematical entities such as sets, lists, maps, relations, functions, differential equations, probabilistic models, and constraints" [6]. Such formalization typically unveils issues not seen otherwise and it moreover fosters an early shared vision among stakeholders. Abstraction plays a crucial role in this process [15,30].

Programs and software are formal entities. The steps from an idea and an informal problem description to a program or a piece of software are the steps from the informal to the formal. Formal methods are thus in the heart of CS.

"'Formal methods' is transforming how Amazon Web Services (AWS) secures the cloud. Security has historically been a manual, high-judgement and thus un-scalable field; automated formal reasoning is challenging that entire structure, changing both the quality of AWS products and the cost structure to support them. The key at AWS has been to avoid 'shiny-object syndrome'."

FORMAL METHODS IN INDUSTRY

While many success stories of applying formal methods in industry concern safety-critical systems [19,25,40], recent literature reports an uptake in the application of formal methods also outside safety-critical applications. For instance, to ensure the quality of cloud services at Amazon [3,32], of cloud databases and weak memory models at Huawei [23,34], and of mobile apps at Facebook [16]. Furthermore, representatives from a wide range of industry sectors have contributed testimonies concerning the use of formal methods in their projects [4]. We include some relevant parts of their contributions here.

Byron Cook (founder of the automated reasoning group at AWS) [4]: "'Formal methods' is transforming how Amazon Web Services (AWS) secures the cloud. Security has historically been a manual, high-judgement and thus un-scalable field; automated formal reasoning is challenging that entire structure, changing both the quality of AWS products and the cost structure to support them. The key at AWS has been to avoid 'shiny-object syndrome' [14] and instead build and apply tools that quietly but reliably change the behavior of engineers. Many leaders at AWS were skeptical of this type of work in 2016, but the success in areas such as cryptography, identity, storage and virtualization has changed minds."

Rod Chapman (AWS) [4]: "In late 2020, AWS announced the availability of strong read-after-write consistency in the S3 storage service. S3 operates at a currently preposterous scale, storing over 100 trillion objects and handling over 10 Million requests per second [39]. Strong consistency ensures that the same view of an object is available to all readers instantly following a write operation to that object. Consistency properties were specified and verified using Dafny [31], a verification-aware programming language."

Ivo ter Horst (ASML) [4]: "To make ASML's lithography systems run reliably and consistently ASML needs software that sends unambiguous instructions in every situation to the carefully engineered hardware. One way that ASML ensures this is by formally verifying (model checking) the specified machine behavior and automatically generating correct and semantically equivalent code from those models" [5].

TEACHING FORMAL METHODS

Fundamental formal methods in CS, comprising modeling, formal specification, refinement, and verification, constitute a key knowledge area with widespread relevance in many of today's innovative applications, like self-driving cars, in a society that increasingly relies on software systems. Currently, discrete mathematics courses are often perceived as early challenges in CS education, disconnected from modern programming languages, yet they are crucial springboards for introducing formal methods. An additional core area directly focused on formal methods could help contextualize discrete mathematics courses for students and could demonstrate why such courses are taught so early as a starting foundation for a CS education.

CS2023 envisions 17 knowledge areas [1], most of which can be enhanced by formal methods (or formal methods thinking). We now discuss some of these knowledge areas to illustrate the strength of formal methods, providing several successful exam-

ples of applications of formal methods in these knowledge areas from the literature, as well as brief suggestions for what to teach in relation with formal methods from [4: Section 4: Educating for Formal Methods in Industry,6,17]. We then conclude this section by providing some supporting evidence from renowned experts in the field for our plea to teach formal methods as part of any CS curriculum since it provides "fundamental Computer Science skills that industry would profit from when hiring computer scientists" [4: Section 4: Educating for Formal Methods in Industry].

ALGORITHMIC FOUNDATIONS

The focus of this knowledge area is on teaching fundamental data structures, classical algorithms, algorithm construction strategies, and computational complexity and computability theory. While there are suggestions on addressing invariants, especially in loops and search algorithms, the CS2023 curriculum lacks explicit competencies related to reasoning about algorithm correctness. "Yet students should learn from the beginning to reason (at least informally) about the correctness of their algorithms" [17]. One way to instill this type of reasoning in students is to teach the classical algorithms with arguments for correctness (e.g., the classical sorting algorithms). For example, the application of formal methods—in particular interactive theorem proving—recently identified a bug in the TimSort sorting algorithm of the Java standard library [26].

ARCHITECTURE AND ORGANIZATION

This knowledge area strives to enhance comprehension of the hardware environments that underpin nearly all computing, and the corresponding interfaces provided to higher software layers. The scope of the hardware considered spans from low-end embedded system processors to high-end enterprise multiprocessors. Teach how formal methods are employed in this area to validate the accuracy of hardware designs and to guarantee that the combination of hardware and software components adheres to their specifications (e.g., to verify security requirements in hardware security architectures [18]). Formal modeling of application architectures by specifying the interface behavior of components is the backbone of architecture design and system integration. This also applies to software architectures with notions such as encapsulation, information hiding, interface modeling, and modularity, which cannot be understood without formal models.

ARTIFICIAL INTELLIGENCE

This knowledge area prepares CS students to recognize when it is suitable to use an AI method (e.g., neural networks, machine learning) and how to apply it, taking the broader societal impacts and implications into account, including issues in AI ethics, fairness, trust, and explainability. Teach how to use formal methods to capture the assumptions of the designs of deep neural networks as used in large language models as well as their verification (with model-checking and interactive theorem proving techniques) or counterexample-based retraining [8,29,37].

PARALLEL AND DISTRIBUTED COMPUTING

This knowledge area encompasses various topics, ranging from parallelization and dependencies to progress, deadlocks, faults, safety, and liveness. Although formal methods are not explicitly mentioned, the suggested learning outcomes for core CS2023 contain examples like "Write a program that correctly terminates when all of a set of concurrent tasks have completed" [1], which obviously requires knowledge of correctness, termination, and techniques for rigorous reasoning about programs, as well as rigorous semantics of concurrency. "This area states as prerequisites logic, discrete mathematics, foundations of software engineering, but none of them in the current status of the ACM standard provides the ability to be able to understand and justify correctness of computational systems" [17]. Formal methods are mentioned as a means of specifying concurrent implementations (e.g., linearizability), and as a means of formalizing inter-process communication (e.g., using a process algebra such as CSP). This shows that, when systems become complex (as in the case of concurrent and distributed programs), formal methods are unavoidable due to the high likelihood of human error. Teach how to understand and justify the correctness of systems in the presence of the topics addressed in this knowledge area (e.g., program parallelization, atomicity, concurrency, progress, deadlocks, faults, safety, and liveness), which in essence lists formal methods as a prerequisite (viz., logic, discrete mathematics, and software engineering foundations).

SECURITY

This knowledge area focuses on instilling a security mindset in CS students by understanding vulnerabilities of—and threats against—software systems, ensuring that security (including concepts like privacy and cryptography) is inherent in all their output. Teach how formal methods can provide the assurance of security properties in algorithms and protocols, ensuring their resilience against attacks. For example, formal methods such as Dafny are used at Amazon Web Services to reason about encryption properties and cryptography infrastructures [11,13].

SOFTWARE DEVELOPMENT FUNDAMENTALS

This knowledge area covers fundamental concepts and skills concerning programming, the use of data structures, refinement, and an understanding of how algorithms impact program performance. As mentioned earlier, algorithms should not be detached from reasoning about their correctness, yet formal methods are not mentioned as part of Software Development Fundamentals in CS2023. In general, however, formal methods concepts enter mainstream programming (think of contracts in C++ or mutexes in concurrent programming). We find it hard to imagine their effective use without knowing formal methods. "We teach children counting and algebra before giving them a pocket calculator, for very good reasons" [17]. Teach to reason (at least informally) about the correctness of programs (e.g., by specifying requirements and justifying why these are met by the proposed program). Support on how to integrate formal methods

is available through a recent textbook [33] that is "suitable for advanced undergraduate and graduate courses in software development."

SOFTWARE ENGINEERING

This knowledge area centers on appropriate means for software design, construction, and verification and validation, primarily through testing. A non-core formal methods module suggests learning outcomes like "describe the role formal specification and analysis techniques can play in the development of complex software" [1], while testing is the primary validation technique in other modules. However, formal methods and testing are not mutually exclusive: "Understanding correctness and reasoning about programs can greatly benefit effective testing" [17]. Formal methods tools for static analysis, like Infer, are used by major companies such as Facebook [16]. The role of formal modeling in software engineering has to be emphasized much more explicitly: teach formal methods. The fact that these are non-core in CS2023 suggests that other topics must be displaced to make room for formal methods. However, we argue that formal methods thinking can (and should) be introduced into a software engineering stream in a lightweight manner without such displacement [17]. Support on how to integrate formal methods is available through a recent textbook [36] that is "suitable for graduate and undergraduate courses in software engineering."

FOUNDATIONS OF PROGRAMMING LANGUAGES

This knowledge area provides a basis for understanding the foundations, implementation, and formal description of modern programming languages. "There is an increasing interest in formal methods to prove program correctness and other properties. To support this, increased coverage of topics related to formal methods is included, but all of these topics are identified as non-core" [1]. Indeed, there are non-core knowledge units on formal semantics and formal development methodologies, with learning outcomes like "use proof assisted programming languages to develop fully specified and verified software artifacts" and "discuss when and how formal methods can be effectively used in the development process" [1]. However, formal methods are "at the heart of modern programming languages" [16] and "some programming languages have very powerful type systems, which require a clear and formal understanding of types and of the type checking process, as well as the ability to employ helpful abstraction" [6]: teach formal methods. "The Software Foundations series [38] uses the Coq proof assistant to rigorously describe both the features of the programming languages being developed and the algorithms that are implemented in these languages" [17].

But not only work on programming language requires knowledge in formal methods. This applies also to modeling languages and as well to tool design [7].

RECOMMENDATIONS BY EXPERTS

According to a recent survey involving 130 formal methods experts [24]—including three Turing Award winners [2], all FME Fellowship Award winners [21], and 17 CAV Award winners [9]—the most suitable places for formal methods in a teaching curriculum is "in master courses at the university" or "in bachelor courses at the university," since this is what 80% and 79.2% of the respondents, respectively, answered to the question When and where should formal methods be taught? [24: Section 5]. Even though more than one answer was allowed, apparently many experts believe that formal methods should be taught at undergraduate or graduate level. Also, the situation of formal methods in CS education is currently receiving "not enough attention" or receiving "sufficient attention, but scattered all over," since this is what 50% and 31.5% of the respondents, respectively, answered to the question What is your opinion on the level of importance currently attributed to teaching of formal methods at universities?

Furthermore, the fact that "engineers lack proper training in formal methods" is the key limiting factor for a wider adoption of formal methods by industry, according to 71.5% of the respondents [24 Section 5]. The survey concludes that "the current situation [of formal methods education] is very heterogeneous across universities, and many experts call for a standardization of university curricula with respect to formal methods," which is confirmed by a recent white paper advocating "the inclusion of a compulsory formal methods course in computer science and software engineering curricula" based on the observation that "there is a lack of computer science graduates who are qualified to apply formal methods in industry" [10], and by the aforementioned recent textbook on formal methods in software engineering [36], which claims that "in computer science and software engineering education, formal methods usually play a minor role only." In the specific context of safety- and mission-critical applications, a very recent paper moreover recognizes "an urgent need to emphasize and integrate formal methods into the undergraduate curriculum in CS in the United States," since "the lack of a well-structured exposure to formal methods is a serious shortcoming in our computing curricula" [35].

HOW TO INTEGRATE FORMAL METHODS INTO CS CURRICULA

The integration of additional material into curricula is always a challenge. Typically, every lecturer is engaged to bring in as much as possible of his or her subject considered highly relevant. How could there be more space for formal methods?

To understand how to bring in additional material covering the subject of formal methods, we have to keep in mind that in every CS subject there is a substantial amount of formalization and thus of formal methods. Classical examples are relational databases or process models. For quite a number of knowledge areas, similar formal methods are used and taught. It would therefore be more appropriate to teach those formal

methods in foundational lectures and only refer to them and use them in the different knowledge areas without having to go deeply into separate introductions. An additional positive effect would be that students will observe these overlaps among the different knowledge areas, thus achieving a clearer understanding of basic concepts and methods of the specific disciplines.

The fact that formal methods are non-core in CS2023 does not imply that other topics must be displaced to make room for formal methods. Formal methods can (and should) be included in any CS curriculum without such displacement. Introducing formal methods as a distinct knowledge area would avoid the need to treat basic formal methods in elective topics in distinct knowledge areas by referring to the knowledge area on formal methods. This would save time in the respective knowledge areas while students understand that basic formal methods are useful in several knowledge areas.

A revision of the ACM CS2023 curriculum is required to adjust the way CS is taught, incorporating formal methods as a knowledge area and a key foundation without displacing other widely accepted engineering aspects of CS. We put forward to what extent eight of the 17 knowledge areas of CS2023 are, or rather should be, related to formal methods.

CONCLUSION

We have presented arguments from three white papers that indicate the indispensable role of formal methods in CS education. Currently, formal methods do not appear as a knowledge area in the ACM CS2023 curriculum. The current offering of formal methods in the CS education is inadequate. We believe that every CS graduate needs to have an education in formal methods, since they can support algorithmic problem solving, AI, model-driven engineering, security, and additional areas of CS, and are moreover applicable in numerous industrial domains, not limited to safety-critical applications. Additionally, not only does formal methods pertain to AI as outlined here, formally verified software itself stands in necessary contrast to the wave of software created by generative AI, offering hard guarantees about correctness that are not possible with statistical approaches to software construction. A revision of the ACM CS2023 curriculum is required to adjust the way CS is taught, incorporating formal methods as a knowledge area and a key foundation without displacing other widely accepted engineering aspects of CS. We put forward to what extent eight of the 17 knowledge areas of CS2023 are, or rather should be, related to formal methods. Three white papers [4,6,17] reinforce the conclusion of an earlier white paper [10]—all together authored by more than 50 computer scientists and practitioners worldwide—in which it is argued that "formal methods need to be better rooted in higher education curricula for computer science and software engineering programmes of study."

In the words of The White House [40: Part II: Securing the Building Blocks of Cyberspace—Formal Methods]: "Given the complexities of code, testing is a necessary but insufficient step in the development process to fully reduce vulnerabilities at scale. If correctness is defined as the ability of a piece of software to meet a specific security requirement, then it is possible to demonstrate correctness using mathematical techniques called formal methods. These techniques, often used to prove a range of software outcomes, can also be used in a cybersecurity context and are viable even in complex environments like space. While formal methods have been studied for decades, their deployment remains limited; further innovation in approaches to make formal methods widely accessible is vital to accelerate broad adoption. Doing so enables formal methods to serve as another powerful tool to give software developers greater assurance that entire classes of vulnerabilities, even beyond memory safety bugs, are absent."

We acknowledge the enormous effort that went into CS2023, but unfortunately without a professional integration of formal methods the result requires further improvement and discussion. More detailed work is needed—going beyond what we can achieve by our arguments here—to show how a core of formal methods is needed, introducing key formal methods that then are used in the various knowledge areas. ❖

Acknowledgments

We thank our co-authors of the three white papers [4,6,17] for their contributions: Achim Brucker, Rod Chapman, Rance Cleaveland, Catherine Dubois, Alessandro Fantechi, Hubert Garavel, Mario Gleirscher, Rong Gu, Stefan Hallerstede, Klaus Havelund, Eric Hehner, Ivo ter Horst, Jeroen Keiren, Markus Alexander Kuppe, Thierry Lecomte, Michael Leuschel, Alexandra Mendes, Carroll Morgan, Peter Müller, André Platzer, Leila Ribeiro, Jan Ringert, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Alexandra Silva, Graeme Smith, Allison Sullivan, Martyn Thomas, Erik de Vink, Tim Willemse, and Lijun Zhang. We also thank Ana Cavalcanti, Cliff Jones, and Luigia Petre for comments on an early draft which helped improve the article (and Luigia also for Figure 2). Finally, we thank the three anonymous reviewers, whose comments have allowed us to improve this article.

References

1. ACM CS2023 Knowledge Areas; https://csed.acm.org/knowledge-areas/. Accessed 2024 Apr 29.
2. ACM Turing Award winners; https://amturing.acm.org/byyear.cfm. Accessed 2024 Apr 29.
3. Backes, J., Bolignano, P., Cook, B., Gacek, A., Luckow, K. S., Rungta, N., Schäf, M., Schlesinger, C., Tanash, R., Varming, C., and Whalen, M. W. One-Click Formal Methods. *IEEE Softw*. 36, 6 (2019), 61–65; https://doi.org/10.1109/MS.2019.2930609
4. ter Beek, M. H., Chapman, R., Cleaveland, R., Garavel, H., Gu, R., ter Horst, I., Keiren, J. J. A., Lecomte, T., Leuschel, M., Rozier, K. Y., Sampaio, A., Seceleanu, C., Thomas, M., Willemse, T. A. C., and Zhang, L. *Formal Methods in Industry. Form. Asp. Comput.* (2024); https://doi.org/10.1145/3689374

5. Binns, L. By computers, for computers: Improving scanner metrology software with generated code; https://www.linkedin.com/pulse/computers-improving-scanner-metrology-software-code-lewis/. Accessed 2024 Apr 29.
6. Broy, M., Brucker, A., Fantechi, A., Gleirscher, M., Havelund, K., Kuppe, M. A., Mendes, A., Platzer, A., Ringert, J., and Sullivan, A. Does Every Computer Scientist Need to Know Formal Methods? *Form. Asp. Comput*. (2024); https://doi.org/10.1145/3670795.
7. Broy, M. and Rumpe, B. Development Use Cases for Semantics-Driven Modeling Languages. *Commun. ACM* 66, 5 (2023), 62–71; https://doi.org/10.1145/3569927.
8. Brucker, A. D. and Stell, A. Verifying Feedforward Neural Networks for Classification in Isabelle/HOL. In *Proceedings of the 25th International Symposium on Formal Methods (FM'23) (Lecture Notes in Computer Science*, Vol. 14000), M. Chechik, J.-P. Katoen, and M. Leucker (Eds.). Springer, Germany, 427–444; https://doi.org/10.1007/978-3-031-27481-7_24.
9. CAV Award winners; http://i-cav.org/cav-award/. Accessed 2024 Jul 29.
10. Cerone, A., Roggenbach, M., Davenport, J., Denner, C., Farrell, M., Haveraaen, M., Moller, F., Körner, P., Krings, S., Ölveczky, P. C., Schlingloff, B.-H., Shilov, N., and Zhumagambetov, R. Rooting Formal Methods Within Higher Education Curricula for Computer Science and Software Engineering – A White Paper. In *Revised Selected Papers of the 1st International Workshop on Formal Methods – Fun for Everybody (FMFun'19) (Communications in Computer and Information Science*, Vol. 1301), A. Cerone and M. Roggenbach (Eds.). Springer, Germany, 1–26; https://doi.org/10.1007/978-3-030-71374-4_1.
11. Chakarov, A., Fedchin, A., Rakamaric, Z., and Rungta, N. Better Counterexamples for Dafny. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'22) (Lecture Notes in Computer Science*, Vol. 13243), D. Fisman and G. Rosu (Eds.). Springer, Germany, 404–411; https://doi.org/10.1007/978-3-030-99524-9_23.
12. Clarke, E. M., Wing, J. M., et al. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv*. 28, 4 (1996), 626–643; https://doi.org/10.1145/242223.242257.
13. Cook, B. Formal Reasoning About the Security of Amazon Web Services. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV'18) (Lecture Notes in Computer Science*, Vol. 10982), H. Chockler and G. Weissenbacher (Eds.). Springer, Germany, 38–47; https://doi.org/10.1007/978-3-319-96145-3_3.
14. Cook, J. Shiny Object Syndrome: The Biggest Problem For Today's Entrepreneurs. *Forbes* (2023); https://www.forbes.com/sites/jodiecook/2023/02/20/shiny-object-syndrome-the-biggest-problem-for-todays-entrepreneurs/?sh=5a90cb4b6709. Accessed 2024 Jul 29.
15. Dijkstra, E. W. On the Cruelty of Really Teaching Computing Science. *Commun. ACM* 32, 12 (1989), 1398–1404; https://doi.org/10.1145/76380.76381 [This is Dijkstra's Contribution to "A Debate on Teaching Computing Science" by P. J. Denning, pp. 1397–1414.]
16. Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70; https://doi.org/10.1145/3338112.
17. Dongol B., Dubois C., Hallerstede S., Hehner E., Morgan C., Müller P., Ribeiro L., Silva A., Smith G., and de Vink E. 2024. On Formal Methods Thinking in Computer Science Education. Form. *Asp. Comput*. (2024); https://doi.org/10.1145/3670419.
18. Ferraiuolo, A., Xu, R., Zhang, D., Myers, A. C., and Suh, G. E. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, Y. Chen, O. Temam, and J. Carter (Eds.). ACM, USA, 555–568; https://doi.org/10.1145/3037697.3037739.
19. Ferrari, A., and ter Beek, M. H. Formal Methods in Railways: a Systematic Mapping Study. *ACM Comput. Surv*. 55, 4 (2023), 69:1–69:37; https://doi.org/10.1145/3520480.
20. FME (Formal Methods Europe); https://fmeurope.org. Accessed 2024 Apr 29.
21. FME Fellowship Award winners; https://www.fmeurope.org/awards/. Accessed 2024 Apr 29.
22. FME Formal Methods Teaching Committee. FME Education Course Database; https://fme-teaching.github.io/courses/. Accessed 2024 Sep 24.
23. Gao, S., Zhan, B., Liu, D., Sun, X., Zhi, Y., Jansen, D. N., and Zhang, L. Formal Verification of Consensus in the Taurus Distributed Database. In *Proceedings of the 24th International Symposium on Formal Methods (FM'21) (Lecture Notes in Computer Science*, Vol. 13047), M. Huisman, C. S. Pasareanu, and N. Zhan (Eds.). Springer, Germany, 741–751; https://doi.org/10.1007/978-3-030-90870-6_42.
24. Garavel, H., ter Beek, M. H., and van de Pol, J. The 2020 Expert Survey on Formal Methods. In *Proceedings of the 25th International Conference on Formal Methods for Industrial Critical Systems (FMICS'20) (Lecture Notes in Computer Science*, Vol. 12327), M. H. ter Beek and D. Ničković (Eds.). Springer, Germany, 3–69; https://doi.org/10.1007/978-3-030-58298-2_1.
25. Gleirscher, M., and Marmsoler, D. Formal Methods in Dependable Systems Engineering: A Survey of Professionals from Europe and North America. *Empir. Softw. Eng*. 25, 6 (2020), 4473–4546; https://doi.org/10.1007/s10664-020-09836-5
26. de Gouw, S., de Boer, F. S., Bubel, R., Hähnle, R., Rot, J., and Steinhöfel, D. Verifying OpenJDK's Sort Method for Generic Collections. *J. Autom. Reason*. 62, 1 (2019), 93–126; https://doi.org/10.1007/s10817-017-9426-4.
27. IEEE Computer Society. Software Engineering Models and Methods; https://www.computer.org/resources/software-engineering-models. Accessed 2024 Sep 24.
28. Jones, C. B., Jackson, D., and Wing, J. Formal Methods Light. *IEEE Comput*. 29, 4 (1996), 20–22; https://doi.org/10.1109/MC.1996.10038.
29. Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, S., Zeljic, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. W. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV'19) (Lecture Notes in Computer Science*, Vol. 11561), I. Dillig and S. Tasiran (Eds.). Springer, Germany, 443–452; https://doi.org/10.1007/978-3-030-25540-4_26.
30. Kramer, J. Is Abstraction the Key to Computing? *Commun. ACM* 50, 4 (2007), 36–42; https://doi.org/10.1145/1232743.1232745.
31. Leino, K. R. M. *Program Proofs*. MIT Press, USA, 2023; https://mitpress.mit.edu/9780262546232/program-proofs/. Accessed 2024 Sep 27.
32. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (2015), 66–73; https://doi.org/10.1145/2699417.
33. Nielson, F., and Nielson, H. R. *Formal Methods: An Appetizer*. Springer, Germany, 2019; https://doi.org/10.1007/978-3-030-05156-3.
34. Oberhauser, J., de Lima Chehab, R. L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., and Vafeiadis, V. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM, USA, 530–545; https://doi.org/10.1145/3445814.3446748.
35. Ramnath, S., and Walk, S. Structuring Formal Methods into the Undergraduate Computer Science Curriculum. In *Proceedings of the 16th International NASA Formal Methods Symposium (NFM'24) (Lecture Notes in Computer Science*, Vol. 14627), N. Benz, D. Gopinath, and N. Shi (Eds.). Springer, Germany, 399–405; https://doi.org/10.1007/978-3-031-60698-4_24.
36. Roggenbach, M., Cerone, A., Schlingloff, B.-H., Schneider, G., and Shaikh, S. A. Formal Methods for Software Engineering: Languages, Methods, Application Domains. Springer, Germany, 2022; https://doi.org/10.1007/978-3-030-38800-3.
37. Seshia, S. A., Desai, A., Dreossi, T., Fremont, D. J., Ghosh, S., Kim, E., Shivakumar, S., Vazquez-Chanlatte, M., and Yue, X. Formal Specification for Deep Neural Networks. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA'18) (Lecture Notes in Computer Science*, Vol. 11138), S. K. Lahiri and C. Wang (Eds.). Springer, Germany, 20–34; https://doi.org/10.1007/978-3-030-01090-4_2.
38. Software Foundations; https://softwarefoundations.cis.upenn.edu/. Accessed 2024 Sep 13.
39. Vogels, W. Diving Deep on S3 Consistency; https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html. Accessed 2024 Jul 29.
40. The White House. Back to the Building Blocks: A Path Toward Secure and Measurable Software. Technical Report. White House Office of the National Cyber Director (ONCD), 2024; https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf. Accessed 2024 Jul 29.
41. Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. Formal methods: Practice and experience. *ACM Comput. Surv*. 41, 4 (2009), 19:1–19:36; https://doi.org/10.1145/1592434.1592436.

Maurice H. ter Beek
CNR–ISTI, Pisa, Italy
maurice.terbeek@isti.cnr.it

Manfred Broy
Technische Universität München, München, Bayern, Germany
broy@in.tum.de

Brijesh Dongol
University of Surrey, Guildford, Surrey, UK.
b.dongol@surrey.ac.uk

DOI: 10.1145/3702231