Does Every Computer Scientist Need to Know Formal Methods?

MANFRED BROY, Informatik, Technische Universität München, Munchen, Germany
ACHIM D. BRUCKER, University of Exeter, Exeter, United Kingdom of Great Britain and Northern Ireland
ALESSANDRO FANTECHI, DINFO, University of Florence School of Engineering, Firenze, Italy
MARIO GLEIRSCHER, University of Bremen, Bremen, Germany
KLAUS HAVELUND, Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, United States
MARKUS ALEXANDER KUPPE, Microsoft Research, Redmond, United States
ALEXANDRA MENDES, INESC TEC, Faculty of Engineering, University of Porto, Porto, Portugal
ANDRÉ PLATZER, Carnegie Mellon University, Pittsburgh, United States
JAN OLIVER RINGERT, Bauhaus University Weimar, Weimar, Germany
ALLISON SULLIVAN, The University of Texas at Arlington, Arlington, United States

We focus on the integration of Formal Methods as mandatory theme in any Computer Science University curriculum. In particular, when considering the ACM Curriculum for Computer Science, the inclusion of Formal Methods as a mandatory Knowledge Area needs arguing for why and how does every computer science graduate benefit from such knowledge. We do not agree with the sentence "While there is a belief that formal methods are important and they are growing in importance, we cannot state that every computer science graduate will need to use formal methods in their career." We argue that formal methods are and have to be an integral part of every computer science curriculum. Just as not all graduates will need to know how to work with databases either, it is still important for students to have a basic understanding of how data is stored and managed efficiently. The same way, students have to understand why and how formal methods work, what their formal background is, and how they are justified. No engineer should be ignorant of the foundations of their subject and the formal methods based on these.

In this article, we aim at highlighting why every computer scientist needs to be familiar with formal methods. We argue that education in formal methods plays a key role by shaping students' programming mindset, fostering an appreciation for underlying principles, and encouraging the practice of thoughtful program

The research performed by Klaus Havelund was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.
Authors' Contact Information: Manfred Broy (Corresponding author), Informatik, Technische Universität München, Munchen, Germany; e-mail: broy@in.tum.de; Achim D. Brucker, University of Exeter, Exeter, Devon, United Kingdom of Great Britain and Northern Ireland; e-mail: a.brucker@exeter.ac.uk; Alessandro Fantechi, University of Florence School of Engineering, Firenze, Toscana, Italy; e-mail: alessandro.fantechi@unifi.it; Mario Gleirscher, University of Bremen, Bremen, Bremen, Germany; e-mail: mario.gleirscher@uni-bremen.de; Klaus Havelund, Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, California, United States; e-mail: klaus.havelund@jpl.nasa.gov; Markus Alexander Kuppe, Microsoft Research, Redmond, Washington, United States; e-mail: makuppe@microsoft.com; Alexandra Mendes, INESC TEC, Faculty of Engineering, University of Porto, Porto, Portugal; e-mail: alexandra@archimendes.com; André Platzer, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States; e-mail: aplatzer@cs.cmu.edu; Jan Oliver Ringert, Bauhaus University Weimar, Weimar, Thüringen, Germany; e-mail: jan.ringert@uni-weimar.de; Allison Sullivan, The University of Texas at Arlington, Arlington, Texas, United States; e-mail: allison.sullivan@uta.edu.

ACM 0934-5043/2024/12-ART6
https://doi.org/10.1145/3670795

design and justification, rather than simply writing programs without reflection and deeper understanding. Since integrating formal methods into the computer science curriculum is not a straightforward process, we explore the additional question: what are the tradeoffs between one dedicated knowledge area of formal methods in a computer science curriculum versus having formal methods scattered across all knowledge areas? Solving problems while designing software and software-intensive systems demands an understanding of what is required, followed by a specification and formalizing a solution in a programming language. How to do this systematically and correctly on solid grounds is exactly supported by formal methods.

CCS Concepts: • **Applied computing → Education;** • **Software and its engineering → Software creation and management**; *Formal software verification;* • **Mathematics of computing**;

Additional Key Words and Phrases: Formal methods, software and systems engineering, computer science university curriculum

ACM Reference Format:
Manfred Broy, Achim D. Brucker, Alessandro Fantechi, Mario Gleirscher, Klaus Havelund, Markus Alexander Kuppe, Alexandra Mendes, André Platzer, Jan Oliver Ringert, and Allison Sullivan. 2024. Does Every Computer Scientist Need to Know Formal Methods?. *Form. Asp. Comput.* 37, 1, Article 6 (December 2024), 17 pages. https://doi.org/10.1145/3670795

1 Introduction

Computer science has to establish itself not only as a powerful engineering discipline, but as a reliable one as well, supported by precise methods that ensure the well-functioning of its systems, keeping the discipline from causing considerable damage and/or loss of trust from the public. Formal methods help to achieve such reliability. In this article, we argue that computer science and, in particular, the science of data and software, can be understood as the "engineering discipline of logic". With this view, data and programs are formal objects, logical entities described by formal techniques such as formal languages with formal semantics. Since digital data and computer programs are formal objects, the ultimate way of dealing with them is using formal techniques.

Any reputable curriculum in computer science has to include teaching some formal foundations, such as logics and formal languages, but formal methods are systematically prescribed not in all curricula. In this article, our goal is to show convincingly that formal methods constitute an integral element of computer science and have to be part of a standard computer science curriculum. This includes teaching what it means to work with formal methods in terms of specifying and designing software systems, implementing them, and proving properties about them.

A fundamental theme that arises throughout this article is that engineering software always includes a step from the informal to the formal, which further illustrates the importance of incorporating formal methods into any computer science curriculum. To help facilitate adoption of this integration, we compare two different approaches: creating a dedicated knowledge area versus enhancing existing knowledge areas by their specific formal methods. The aim of this article is not to provide an exhaustive literature review: we use citations parsimoniously only to support our arguments, with no claim that such references fully represent the vast literature about formal methods and their applications.

The article is organized as follows. Section 2 overviews the mathematical background for formal methods. Section 3 provides a description of what a formal method is. Section 4 discusses the use of formal methods in some computer science sub-disciplines. Section 5 addresses the role of formal methods in engineering and consequently in industry. Section 6 discusses the integration of formal methods in computer science curricula. Finally, Section 7 closes the article by some concluding remarks. We do not give a separate section on related work, but include all references to related work into the general discussion wherever appropriate. Although this article was triggered by the

ACM computer science curricula recommendations, we do not discuss these recommendations in detail, but rather show why formal methods need to be integrated explicitly into computer science curricula. We address, in particular, young lecturers in computer science, giving hints on how they may improve their lectures by a solid integration of formal methods.

2 Mathematical Background

Mathematics, and in particular discrete mathematics and logic, is the theoretical basis of computer science. In particular, the foundation of formal methods is the mathematics used to give semantics to these methods and to inspire their formalisms. As such, it is commonly used to explain computer science concepts, for example, in terms of type systems and semantics, whether static (such as abstract interpretation) or dynamic (such as axiomatic, denotational, or operational semantics). There is a body of formal, mathematically based foundations of computer science that forms the subject of the theoretical computer science field. This includes for example, set theory, logic, formal languages, transition systems, automata theory, data structures, algorithms, computability, computational complexity, semantics of programming languages, logics of programming, the theory of concurrency, probabilistic computation, cryptography, and machine learning. Moreover, the foundation also includes topics from continuous mathematics, such as differential equations for cyber-physical systems modeling and quantitative analysis, or topics from probability, optimization theory and linear algebra for artificial intelligence (AI).

In addition, there are specific formal foundations in subfields of computer science, for instance, relational models in databases, or the logical theory of binary circuits in computer architectures. Such formal foundations consist of theories, concepts, and structures in computer science without necessarily relating them to engineering methods. Formal methods themselves employ these formal foundations, for example, propositional logic for analyzing software configurations and software product lines, or interactive provers based on temporal logics for showing general safety and liveness properties. This way, formal methods bring to bear the mathematics and logic needed for supporting, or even guiding, the rigorous tool-based development or maintenance of critical pieces of software, electronic hardware, or similar systems [15].

3 What is a Formal Method?

This section addresses the question of what constitutes a formal method. We first address the question broadly, defining the core characteristics of a formal method (Section 3.1). We then discuss what formal methods thinking is, suggesting that it goes beyond just applying formal methods (Section 3.2). Finally, we consider the role of formal methods in the software development process (Section 3.3).

3.1 The Core Characteristics of a Formal Method

The word "method" literally means a systematic pursuit of knowledge investigation and a certain mode of proceeding when solving a problem. In recent centuries, a method has been described as a process for completing a task, which means that methods have something in common with algorithms. However, in contrast to algorithms where all steps are defined precisely such that the algorithm can be "executed" completely schematically without additional inventiveness, in general, the application of a method requires human creativity and expertise. Nonetheless, methods have with the notion of algorithm in common that their purpose, result and effort are all aimed at solving a problem or pursuing a task. Roughly, we may classify methods with respect to their degree of formality as follows:

— Informal methods: informally described methods that can be applied without using formalisms (example: "informal code review"),

— Semi-formal methods: methods using formalisms and formal artifacts, such as for example formal syntax or graphics, in combination with informal steps (example: "informal code review on the basis of formal assertions"),
— Formal methods: methods using formally defined steps on formal artifacts (example: "formal specification of a program and proof that a program satisfies the specification") with proven guarantees provided the method is successfully and properly applied.

With a formal method, one can provide a provable formal chain of arguments that provides evidence for the correctness of claims. There seems to be no doubt regarding the meaning of "formal" in "formal methods": a solid and rigorous mathematical and logical argument based on a carefully worked out theory. Equally important is the "methods" part of formal methods: systematic procedures for purposeful and pragmatic applications following clear goals leaving room for and requiring creativity. Examples that do not qualify as formal methods based on this understanding are, for example, informal software process models, which lack required mathematical foundations, as well as formal theories, such as lambda calculus, that do not directly provide methods.

Formal methods have multiple characterizations in the literature (see [1, 5, 28]) as languages and techniques with rigorous mathematical foundations. In addition, many formal methods are automated and supported by a number of tools such as model checkers and theorem provers. For example, in model checking, transition systems and temporal logics represent the mathematical theories ("formal" part) for modeling of reactive systems, while the specification and analysis of safety properties are supported by systematic procedures and techniques ("methods" part).

Formal methods exist in many engineering areas (e.g., mechanical or electrical engineering) not just in computer science. However, in computer science, there is a rich field of application areas (e.g., assurance of critical software and hardware) where formal methods are exceedingly useful. This is a consequence of the fact that computer science is a discipline with formal objects as its subject.

Formal methods are rigorous. They guarantee to achieve results justified by formal theories. Applying theories for creating a proof requires some understanding of the formalisms and the theory, and some discipline to follow the rules and creativity to apply these in a way that leads to the intended results. In fact, proofs provide fundamental insights, particularly, into the behavior of a system. Interesting examples are invariants for state-oriented programs, which characterize what properties of a state are maintained when the program is executed. Loop invariants are used to deal with iteration in computing. Invariant proofs of program correctness can be rigorously formalized. Here, the difference between formal foundations and formal methods becomes apparent. State machines can be used as a formal foundation of invariants in computations. Invariants are defined as predicates that hold for sets of reachable states. However, loop invariants, in general, cannot be generated in a completely automatic way, their formulation requires human creativity. Even though the concept of invariants does not require a lot of formalism beyond the programming language, in order to understand why invariants help and how we can profit from using invariants, it is crucial to understand the theory behind invariants. Most important, by the theory of invariants it is guaranteed that if an invariant holds at the beginning of a loop, then it holds throughout the loop and after it has terminated.

Of the range of formal methods that have emerged over time, some focus more on automation and others interpret the term "methods" in an engineering-oriented way, guiding software or systems engineers through the construction steps they typically perform to arrive reliably at a suitable solution or to transform a preliminary solution into a better or more appropriate one. Hence, formal methods come in many shapes and sizes, from lightweight automated analysis engines that check if a specific hard-coded property is true about a system to more heavyweight interactive

theorem provers that handle complex properties and need human guidance to help complete the proof. In either case, the formal method technique is helping to provide the user with a guarantee of correctness of the design or implementation of their system with respect to some explicit or implicit specification. An example of lightweight verification with an implicit specification is an automated correctness check of null-pointer dereferencing used in almost all of today's integrated development environments, which is helpful, even when this can neither be checked fully automatically nor establishes all desirable properties of a program. An example of heavyweight verification with explicit specifications are security proofs showing the absence of side-channel attacks in cryptographic algorithms.[1] Tools are and can be built based on formal methods supporting their application. However, for a professional use of such tools, the understanding of formal methods is indispensable.

Regarding explicit specifications, the typical interfaces to the users of formal methods are specification languages to formulate abstractions. These languages may range from intuitive visual representations through textual domain-specific languages to rich higher-order and modal logics. Unsurprisingly, the pragmatic aspects of formal methods also range from checking product configurations for consistency of multiple views on software architecture to synthesizing code for efficient parallel execution of complex system behaviors. This range of formal methods across all levels opens opportunities to embed these methods into daily software-engineering practice.

The wide spectrum of useful applications of formal methods – as well as capabilities needed to apply them, including understanding their theoretical foundations, grasping how certain tasks can be supported by formal theories and knowing how a method can be applied to achieve a task is evidenced by Garavel et al. [13], where opinions and position statements were collected from more than one hundred experts in the field.

3.2 Formal Methods Thinking

A fundamental aspect of formal methods is the underlying approach to think about a problem and its solutions. Programmers typically apply some lightweight formal methods and associated thinking in their daily work. As an example, a very common use of formal methods by programmers is that of type systems, which support defining formal requirements on value expressions and then checking that the expressions produce values of the given types. As a syntactic method for enforcing levels of abstraction in programs [24], type systems also aid programmers in decomposing and structuring their programs and allow programmers to reason using high-level abstractions. 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.

In this process, abstraction is key [20]. As stated by Guttag [17]: "The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context". The complexity of computer systems can be approached only through a collection of abstraction layers. In every aspect of software development, we use abstraction: we abstract from actual machines when programming with a high-level language; we abstract from the underlying network layers when we program a distributed web service; we abstract in representative test data all the possible inputs to a function when we test it; we abstract from the implementation of a library function when we call its interface. Every time we make such abstractions, we make an act of faith: we postulate that the abstracted underlying layers behave as expected and that the compiler does not introduce bugs. To transform our faith into certainty,

[1] For a definition of: lightweight and heavyweight interpretations of the perimeter and scope of formal methods we refer to [13].

the appropriate way is to formally prove that the underlying layers abide by a contract that can be trusted at the higher levels. Building on "acts of faith" may often seem unavoidable in practice, due to cost, complexity or even undecidability constraints regarding formal methods applications; but even then formal thinking gives awareness that such "acts of faith" have a solid background and can become "acts of certainty".

Mathematics is a powerful abstraction "tool", which can be applied to understand, design, and reason about software applications and systems in general. Formal methods provide a realization of these insights as they bridge the gap between pure mathematics and its application to software development. Formal methods thinking (see [39]) 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.

Some formal methods focus on the description and analysis of concurrent systems, providing useful abstract notations and concepts to think in terms of, without being bogged down by implementation details. With these concepts, an application can often be described very succinctly in the initial phases of its development. The act of just writing down a mathematical formalization can reveal many issues and bring a group of people to an early shared vision.

Consider the example of design by contract [21], which is increasingly supported in programming languages (e.g., Java/JML, Dafny). In the design-by-contract approach, software systems are viewed as families of software components that interact with each other according to precisely defined interface specifications of client-supplier obligations, referred to as contracts. Logical contracts are a key concept in formal methods, where the behavior of software is specified using preconditions, invariants, and postconditions. Preconditions specify the conditions that must be met when a function or method is called for it to operate as expected, while postconditions specify the conditions that must hold true when the function or method call terminates. Invariants (see Section 3.1) specify properties that are maintained from entry to exit and, thus, constitute some of the most important aspects of the state that the rest of the program can rely on. Thinking about programs in terms of such logical contracts instills in a programmer a mentality of "think about design first, program later", allowing for the detection of design errors earlier in the process and forcing them to reflect and specify precisely what the intended behavior of the components is. This way of thinking supports modularity, encapsulation, information hiding, abstraction, problem decomposition, structured design, and the recognition of patterns in the problem, allowing for better code and systematic reuse. The contracts are then the specifications for the implementation of the components.

3.3 Formal Methods as Part of the Software Development Process

For software engineering, formal methods are used to specify and reason directly about the artifacts (e.g., programs) of concern since the artifacts themselves can be considered as formal objects. For example, reasoning about a program and proving that it satisfies a formal specification can be done all within one logical system. This is in contrast to traditional engineering disciplines, where the artifact being specified and reasoned about usually is of a physical nature and very far removed from its formalization (e.g., a building versus the equations that describe its structural characteristics). Notably, also compilers or machine designs, which – in case of being faulty – could undermine proofs, are artifacts that can be developed formally. Even more important than the ability to reason about a program after creation is the way in which formal foundations enable a design process in which correctness and verification play an integral part.

Unfortunately, creating systems as well as software that do not completely comply with the final client's wishes is a common experience for software engineers: the blame and the burden of improvement of software is put on software engineers and programmers. But in many such

cases, the client's wishes are not stated precisely, or are ambiguous, or even missing, hence the concept of correctness becomes questionable and sometimes questioned in legal disputes. Plenty of evidence for software problems and disputes can, for example, be found via the RISKS Digest.[2] Such disputes can be mitigated by having appropriate models and precise specifications. The survey in [13] provides anecdotes on the use and potential of formal methods for this purpose.

Fundamental concepts in software engineering are requirements, specification, implementation, validation, and verification, all of which are formal objects and all of which may concern formal objects that have representations within the field of formal methods:

- Identifying Requirements: When developing and evolving software and software-intensive systems it is challenging to come up with a valid understanding of the requirements. Different and potentially conflicting expectations of various stakeholders must be considered in a specification: developing this understanding is always a compromise. Requirements are usually stated informally, but it is advantageous being capable to formalize requirements to avoid ambiguity.
- Modeling and Specification: The development of a formal specification must strike a balance between stakeholder expectations. A sufficiently formal specification can make intensions clear and avoid misunderstandings. For instance, a verification method can provide the missing assumptions needed to prove the implementation correct. Naturally, specifications do not only contain functional requirements, but also non-functional aspects.
- Design and Implementation: Data and executable programs are developed according to the given specification, passing from the high level of abstraction of the specification to the concrete level of software architectures and programs.
- Validation: The validity of requirements with respect to the user's needs has to be confirmed. Since requirements are always the result of compromises between stakeholders, validation is most often challenging, if not supported by a shared formal understanding.
- Verification: The correctness of the implementation with respect to the specification has to be shown. As soon as a specification is formal, and the implementation is formulated in a formal language with a formal semantics, the question of correctness translates to whether an implementation satisfies the specification.
- Evolution: In software maintenance code is changed. There it is of high importance whether local changes have global effects and which additional parts have to be changed, too.

Software is correct if it respects all its requirements, functional and non-functional. Correctness can only be claimed in reference to specifications. That is, correctness can be ascertained only by a comparison between the actual program behavior and the one expected according to the specification.

A crucial point in any specification is to be precise about assumptions about the environment in which the created artifact will be deployed. A recurrent problem with specifications written in natural language is glossing over such assumptions. An enormously expensive and damaging example of missed assumptions is the security flaws that are costing computer users billions of dollars annually. Again, see the RISKS Digest for a plethora of anecdotes about software flaws, their costs and causes, as collected by the community over the past four decades. Proper use of formalism can identify which assumptions are required to meet a specification. For instance, a verification method can provide the missing assumptions needed to prove the implementation correct. Careful documentation of assumptions, for example, as done in contract-based development [21], is also

[2] http://catless.ncl.ac.uk/Risks/

key to the continued use and change of large software products because the assumptions act as a warning against modifications that could lead to disaster.

In practice, testing is typically used to detect faults in software, but testing requires specifications and architecture modeling to reliably detect undesirable behavior. For testing, the comparison is made between the results given by a program that is executed over a finite set of test data and the expected output according to the specification for that test data. Testing can almost never be exhaustive and to estimate to which extent testing can help is again a topic for formal methods and can be done for example by measuring coverage of a formal model of the software, used for test case generation.

Formal development and verification techniques can provide justified confidence on any desired notion of correctness that can be expected by the produced software. Standards like EN50128 [29], ISO 26262 [31], or Common Criteria [32] strongly recommend formal methods in critical applications.

4 Formal Methods in Computer Science Sub Disciplines

In this section we discuss the use and importance of formal methods for various areas of computer science. We first provide an overview of relevant disciplines (Section 4.1). We then draw out two particular disciplines, which are currently receiving a lot of interest from academia as well as industry, namely cyber-physical systems (Section 4.2), and AI (Section 4.3).

4.1 Formal Methods Spread Over Many Knowledge Areas

Beyond the need for a knowledge area of formal methods in computer science education, as discussed in the previous section, there is a need for teaching formal methods in various specialized domains (see [41]). As systems and software become more complex, interconnected, and ubiquitous, the need for rigorous methods to ensure software correctness and therefore systems reliability increases. Examples of key areas in computer science in which formal methods are indispensable include:

- Algorithmic Foundations, where formal methods are used to analyze the correctness and efficiency of algorithms and data structures. An example includes the verification of the TimSort sorting algorithm of the Java standard library using KeY, and the discovery of a bug in the algorithm as a result of this verification effort [7].
- Architecture and Organization, where formal methods are used to verify the correctness of hardware designs and to ensure that the integration of hardware and software components meets their specifications. An example is the verification of security requirements of complex hardware security architectures [10].
- Artificial Intelligence, where formal methods are used, for instance, in deep neural networks (DNNs) for verification and for retraining by using counter examples and also to capture rigorously the assumptions made during their design [27]. The use of DNNs in large language models is attracting a lot of attention and promises to revolutionize the way people interact with computers; the need for formal methods certifications for DNNs is thus essential.
- Security, where formal methods can be used to guarantee security requirements for algorithms and protocols, ensuring that they are secure and resistant to attacks. A recent example in industry is the use of formal methods at Amazon Web Services to prove properties about encryption, which highlights the power of formal methods in practice [6].
- Software Engineering, where formal methods, as already mentioned, can be used in several development lifecycle phases including system specification and its validation alongside or

in place of testing [22]. Further examples are static analysis tools such as Infer [8] which is used by well-known companies such as Facebook, Amazon Web Services, Microsoft, Mozilla, Uber, WhatsApp, and many others.
— Databases, where formal methods can be used for knowledge representation and for reasoning about data consistency. For example, Description Logic allows the rigorous analysis of database schemas [43]. In the treatment of and the search in large data sets, such abstractions and formal reasoning are highly relevant.

When learning about formal methods, students get access to rigorous and systematic approaches for providing formal guarantees on the behavior of algorithms and systems, ensuring, respectively, their correctness and reliability, something that is undoubtedly in high demand across all knowledge areas of computer science.

It is impossible to talk here about all computer science sub-disciplines and the way formal methods are helpful there. Therefore, we choose only two examples of highly relevant sub-disciplines to demonstrate the significance of formal methods for them.

4.2 Formal Methods for Cyber Physical Systems

We have already mentioned the general use of formal methods in software engineering. A special area where there are good reasons to integrate aspects of formal methods into computer science curricula is that of cyber-physical systems, where computer programs interact with the physical world. As a representative of similar courses taught in other universities, we take for example the Logical Foundations of Cyber-Physical Systems (LFCPS) course [25] at Carnegie Mellon University, that teaches the foundations of cyber-physical systems, that is, systems that combine computer control with physical systems as in robots or aircrafts; this simultaneously serves as a first course on logic and formal methods. The course intentionally focuses on the heart of the matter, cyber-physical systems design, right away, bringing in the required background and formal methods aspects as much as needed.

While this requires significant reorganization of materials (compared to a conventional linear presentation of the background of cyber-physical systems), the big advantage is a clear motivation of taught material by practical applications; this serves as a guiding motivation for the need of formal methods. In the LFCPS course, students also experience the difference between specification and verification by first informally developing robot controllers that they only specify and of which they conjecture correctness (called betabots) while subsequently developing formally verified robot controllers (veribots). What is a particular eye-opener for students in the course is that, despite their best intentions and best practice software development principles, cyber-physical systems have so many subtleties in store that their first designs still have bugs until they are being helped by formal methods.

4.3 Formal Methods and AI

Due to the rapid development within the AI field and its potentially extreme impact on society, it is essential to mention the use of formal methods specifically within this field. AI introduces new important applications and need for formal methods, both when it comes to verifying AI systems and to specifying and verifying software generated by AI. This is essential, not only because such systems are used in manifestly safety- or security-critical applications (e.g., for the detection of traffic signs by semi-autonomous cars [4]), but because they are becoming more and more entangled into every aspect of our lives, with unexpected critical threats in domains beyond security and safety, such as ethics.

Still, formally modeling, specifying, and analyzing such systems is an area that requires novel approaches from scientists and engineers educated in both formal methods and AI, in particular, machine learning (ML). This is mainly due to three reasons. Firstly, the behavior of ML/AI-driven systems is a result of training those systems with data. Hence, the selection of training data plays a crucial role in properties of the final systems. Secondly, these systems do not have human written algorithms that can be analyzed with existing formal methods because they defy standard comprehension. Even the internal representation of a particular learnt scenario is hardly understood. Thirdly, the properties that an ML/AI-driven system should satisfy are only specified to the extent that regulations (e.g., ANSI/UL4600) suggest sufficiently low error margins (e.g., for classifications) to be acceptable. Taking, for example, systems for classifying traffic signs: what does it mean in practice that a system recognizes a STOP sign in 99.99 % of the cases, when one misclassification of a STOP sign can result in an accident endangering the life of humans? Even worse, we currently do not even have good methods for modeling the inputs that are misclassified – and we have seen serious attacks on such models where attackers made small modifications to inputs that resulted in safety-critical misclassifications [9]. We currently see increased activities of the formal methods community to address these challenges, for example, using model-checking techniques [19], and interactive theorem proving [3].

In the emergence of AI for assisting in the development of software, it is foreseeable that some of the traditional programming skills are on the verge of being replaced by AI-based design automation. Indeed, it can be expected that recent AI technologies (e.g., ChatGPT,[3] GitHub Copilot,[4] and other tools going beyond search-based software engineering and traditional program synthesis) will revolutionize software development. Importantly, these technologies will broaden rather than lessen the necessity for software practitioners to properly specify the correctness of the resulting AI systems. Formal methods are well-positioned to play a crucial role in that paradigm shift by making prompt engineering (the process of creating and reviewing high-quality prompts) more formal, and by checking whether the generated code meets the specifications. We thus concur with Greengard's view [16] that practitioners need to adopt a more abstract and rigorous notion of software engineering to reduce potentially subtle but still critical errors in AI-generated programs.

5 Formal Methods in Software and Systems Engineering

In this section, we discuss on the role of formal methods in software and systems engineering in an industrial context. We first discuss their general role in computer science seen as an engineering discipline (Section 5.1). We then consider the role of formal methods as cross cutting between engineering disciplines and stakeholders (Section 5.2).

5.1 Computer Science as an Engineering Discipline

Engineering is the application of scientific methods to solve problems by designing and building systems. Engineers design and construct artifacts, which should be built to serve their intended purpose reliably. In most cases, the created objects should be designed to last and they should be built at a reasonable cost. Software systems, in particular, are among the largest and most complicated artifacts that humans have created [30]. The investment by industry and government in software is enormous, but today software cannot, in general, be considered to meet the highest engineering standards (see [40]).

Engineers of physical artifacts need to learn methods that underpin documentation and reasoning about their designs; no engineer would be allowed to work on a bridge or aircraft design

[3] https://openai.com/blog/chatgpt
[4] https://github.com/features/copilot

without having a grasp of appropriate mathematical concepts. By the same token, software engineers must learn how precise specifications are constructed and how their key design decisions are subject to rigorous justification.

Building reliable systems requires rigorous development approaches based on abstract models, unambiguous specifications of the functional and non-functional requirements, rigorous tests, and verification methods to ensure that the final systems satisfy their requirements. While software systems do not suffer wear and tear, the environments of software systems keep changing, leading to new and updated requirements that ask for long-term maintenance plans. Archetypically, computer science is different from traditional engineering disciplines, in that it fundamentally completely relies on formal concepts and methods.

Education plays a major role in the adoption of formal methods in industry and the lack of a broad education in the field shows in their limited adoption. In many STEM[5] fields (e.g., mechanical, civil, chemical, and electrical engineering) challenges of formalization seem to be less apparent. Decades ago, Tony Hoare stated: "I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law" [18]. The situation motivating his observation has improved significantly since then. Indeed, in critical software domains, such as the railway industry [11, 42], formal methods are used and, for the highest safety integrity levels, even mandated by standards (e.g., EN 50128).

Another historical success story is in hardware verification with, for example, Intel and AMD both using model checking and theorem proving to ensure the correctness of their processors (e.g., resolving the Pentium FDIV bug at Intel, using ACL2 for verifying the K5/K6 processors at AMD). In addition, formal methods have also found success in cyber security, as evidenced by formal methods being a top-level knowledge area in the Cyber Security Body of Knowledge[6] (CyBOK). Cloudflare's use of Tamarin[7] highlights formal methods being successfully applied to industrial security applications. More broadly, formal methods have recently been applied in companies such as ARM (for verifying a security-critical firmware component [12]), Facebook (e.g., using Infer, see [8]), Amazon (for business-critical clouds, see [38]), and for Blockchain technology.

Despite these success stories, instead of a revolution in the integration and use of formal methods in industrial contexts (e.g., cross-disciplinary, model-based, architecture-centric [2]), complex software practice adopts powerful and integrative abstraction techniques rather hesitatingly and severe software-caused problems keep occurring.

We envisage three strategies for transferring formal methods to industry to mitigate the phenomena associated with Hoare's observation and contribute to reducing software-caused errors: (i) making formal methods affordable by automation and integration, (ii) making formal methods more accessible, and (iii) improving formal methods education. These strategies are detailed next.

First, the use of formal methods could be made more affordable, such that the costs are more easily counterbalanced by the need for high quality. For instance, David Parnas suggests quite radically to adopt the way engineers use mathematics for property calculation, resulting in practical formal methods with simpler specification and reasoning facilities [23]. The classical verification problem is addressed by the introduction of automated provers (model checkers/model finders, such as, e.g., SAT and SMT solvers), which perform automated analysis of specifications as well

[5]Science, Technology, Engineering, and Mathematics
[6]https://www.cybok.org/knowledgebase1_1/
[7]https://blog.cloudflare.com/post-quantum-formal-analysis/

as code. Moreover, formal methods are now being incorporated into programming languages. Examples are Dafny [33], Eiffel [21], and SPARK/Ada [34].

Second, formal methods could be made more accessible, by supporting other more pragmatic methods that are more familiar to users, providing semantic underpinning and analysis. As examples, both SAT and SMT have been used to give semantics to SysML (see [35, 36]), and Event-B has been used to support UML (see [37]), including the traditional visualization capabilities of UML, but founded in formal semantics. However, one may argue that engineers using such tools in a professional way should have sufficient understanding of the formal methods behind the tools rather than using tools as just hidden black boxes.

Third, universities could enforce a methodological school on their graduates and, overcoming industrial pragmatics, nurture their (under)graduate formal methods specialists. Such a school would, for example, respond to anticipated needs such as correct-by-construction AI-based software development as well as by discussing current technologies through the eyes of the underlying formal methods concepts. In general, a better and wider formal methods education could not only lower method adoption costs due to availability of skilled personnel but also enlarge the benefits of formal methods to many application domains. We discuss this in more depth in Section 6.

There is of course an investment to be made by anyone who aspires to become a professional engineer: that cost is in time and study. Although structural engineers might build small one-off systems without having to apply the full force of the mathematics they learn at university, faced with a large project that will employ many people and expend large sums of money, professional engineers insist on precise specifications and use all the tools that they have learned to be as certain as possible that their design will yield a satisfactory engineered result.

Fortunately, early computing pioneers realized how important it was to reason about programs: both von Neumann and Turing published papers in the 1940s that showed it was possible to record a proof that a program had its intended effect. Researchers over the intervening decades have increased the tractability of both specification and reasoning and have constructed software tools that support recording proofs. These tools do not remove the need for engineers to understand the formalism; on the contrary, they can only be used effectively by engineers who understand how abstract models can provide precise specifications. Just as structural engineers do not always apply their most rigorous techniques, not all software needs to be developed formally. Nevertheless, for any serious software engineer wanting to apply more rigorous methods, it is essential to be prepared for such applications.

5.2 Formal Methods as Cross Cutting in Engineering

Although computer science is an engineering discipline in its own right, it interacts with other engineering disciplines and gets more and more into the role of a coordinator and controller of the various involved engineering disciplines, with important demands on formality. Computer programs do not run in isolation but interface with the physical world. After all, computers are often intended to impact the real world. This is most obvious in cases where computer programs directly and visibly control electromechanical machines. Going beyond the limits of code reviews (e.g., error proneness, high cost, informal reasoning), even if one can observe the effect that a computer program has on a physical system through standard-compliance tests, trial and error, the resulting understanding is limited to the finite number of cases that were tested. All predictions beyond this negligible experience of finitely many tests out of the infinitely many possible scenarios need formal descriptions of the relevant objects in the physical world as well as specifications of their behavior. This is the basis of system design and its verification.

Creating suitable formal descriptions of the physical world requires several formal methods skills, including (1) abstraction to identify both the relevant part of the physical world and the

relevant level of detail, (2) rigor to obtain unambiguous descriptions, (3) clear understanding of the model's semantics. Justifications of the role and correctness of a computer program that interacts with the physical world require even more formal methods skills related to formal specification, formal design and formal verification, as well as the taming of complexity. When software is used in cyber-physical systems, requirements typically come from domain experts. The communication between the software engineer and the domain expert is very often the reason why requirements are incomplete or wrong: domain experts are not aware of all the implicit assumptions they make and thus they do not explicitly formulate them for the software engineer to model and implement. Formal Methods can help by formalizing requirements and validating them, thus significantly reducing communication gaps.

More generally, formal methods can be used for precise communication between stakeholders working in different areas. As already highlighted in Section 4.1 and in several surveys (e.g., [5, 11, 13, 14]), formal methods have been used across many application domains (either critical or non-critical), computing technologies (e.g., from digital circuits to programming languages), and development stages (e.g., from ideas and expectations to specification and to testing), spanning from academia to industry.

This cross-cutting use demonstrates that rigorous analysis and knowledge transfer is intrinsic to engineering best practices at many development stages and abstraction levels. Although the style of formalization varies across tasks, the necessity of formalization seems to correlate more with the criticality of adequate knowledge transfer than with the type of task. One can argue that this criticality is higher in industry than in academia because of business risks and revenues at stake.

6 Integrating Formal Methods into Computer Science Curricula

As computer science continues to evolve, the integration of formal methods into the curriculum now ensures that graduates are well-prepared to contribute to the software-powered society of tomorrow. This section delves into the importance of incorporating formal methods in computer science curricula, exploring the possibilities of either teaching a dedicated knowledge area for formal methods or integrating them throughout various knowledge areas. It discusses how to include formal methods into computer science education, ensuring that graduates understand the principles underlying their work, even if they may not need to apply these methods in their everyday practice explicitly. By weaving formal methods throughout the curriculum, educators can emphasize the significance of abstraction and formal precision in computer science.

In teaching computer science, formal methods help both as a means for a theoretical understanding in the various subareas, to show the soundness of certain methods, and to manage certain tasks with guaranteed quality. Certainly, students must learn specific formal methods to understand how such methods form a firm basis for software and system design.

This section first discusses formal methods as a knowledge area of computer science education (Section 6.1). Then it is argued how the role of formal methods can be integrated smoothly as part of programming education (Section 6.2). Finally formal methods are discussed as a connecting theme in computer science education (Section 6.3).

6.1 Need for a Knowledge Area on Formal Methods

There is a clear set of fundamental formal methods topics in computer science that forms a knowledge area. This includes the key concepts of formal specification, refinement, and verification. These topics are relevant for many areas in computer science and show up in numerous innovative applications.

Currently, discrete mathematics courses, which are often taken within the first or second year of a computer science bachelor's degree, have the reputation of purposely filtering out weaker

students. The mathematical logic presented seems divorced from modern programming languages. However, dedicated early courses offer a springboard to introduce students to formal methods and their power. They impart to students the significance of getting into the habit of producing software models, and other formal artifacts, as a starting point and guide for programming. A knowledge area directly focused on formal methods can help contextualize discrete mathematics courses for students, and can demonstrate why such courses are taught so early as a starting foundation for a solid computer science education.

6.2 Formal Methods as Part of Programming Classes

Formal methods can naturally be integrated into the documentation given for programming assignments across other knowledge areas within the computer science curriculum. This integration can help alleviate current problems within the computer science education community. Namely, undergraduate students often rush into a programming assignment before understanding what is being asked of them [26]. This often results in students feeling frustrated, as they have wasted their time solving the wrong problem. A further consequence of this behavior is that by solving the wrong problem, students may not be learning the intended lessons for a programming assignment. This can only be mitigated by the adoption of software models and formal specifications in the assignment description.

By their nature, software models force students to slow down and to fully understand their assignment before they start to code. Hence, integrating formal methods into computer science curricula does not need to displace other elements in a computer science curriculum. If taught early, formal methods can instead enhance the experience for students to access knowledge areas more successfully.

6.3 Formal Methods – A Connecting Theme in Computer Science Education

The significance of integrating formal methods as an essential theme into computer science education shows to be obvious. Skills and knowledge acquired from studying formal methods provide a solid foundation that underpins the practice of almost all computer science domains. By understanding and appreciating the principles and techniques of formal methods, students develop an enhanced ability to identify requirements, to formulate specifications, to work out designs, to implement software systems, and to reason about their correctness, reliability, and security, across domains. Formal methods serve as a powerful tool for abstraction and communication, enabling students to understand and to articulate complex ideas better. To summarize, formal methods aid in addressing cross-cutting concerns which promote the maturity of computer science as a scientific engineering discipline.

Undoubtedly, tools have changed the landscape of formal methods and contributed to the successes of formal methods in industry. The availability of tools and automation today is making it significantly easier to integrate formal methods into the curriculum compared to a decade ago. Students learn to use tools and this is a high motivator and catalyst for adopting formal methods.

7 Conclusion

Does every computer scientist need to know formal methods? Our stance, supported by the arguments made throughout this article, is “yes, they do”. Even more, software developers not being aware of the various benefits of formal methods cannot be called computer scientists or software engineers. As we have argued, there is a rich spectrum of formal foundations and formal methods that builds the indispensable backbone of computer science. We have also explained that formalization is at the heart of crafting reliable abstractions, an essential skill of any computer scientist, being capable of reliable communication between stakeholders, building systems reliably

fulfilling critical functional and non-functional requirements (e.g., safety, security), and, finally, formal methods help educating the software practitioner's in forming a critical and rigorous mind. Consequently, it is key to know about, and to understand, the following categories of formal methods and their foundations:

- —Basics: formal foundations of computer science and formal methods, showing how foundations are used to achieve engineering goals, must be taught to all students in the field;
- —a careful selection of formal methods that are fundamental in major fields of computer science, forming an integral part of the knowledge areas, such as, for instance, axiomatic definitions of abstract data types in the field of programming languages or Hoare-style verification techniques must be taught to all students in the field;
- —specialized formal methods designed for solving particular problems for specific fields of computer science should be taught to all students specializing in that field.

When designing curricula in computer science, it is a major task to integrate formal methods as a fundamental theme and to identify which formal methods belong to which categories. This supports the structuring of curricula in a way that formal theories and methods form the underpinning of computer science education providing some blueprint for the various knowledge areas.

To form a scientific curriculum in computer science, it is not enough to informally introduce the subjects of the various knowledge areas. A scientific curriculum needs a structure where solid foundations form the basis for concepts and formal methods which then are then applied in the various knowledge areas. Formal methods help as a connecting theme to relate the separate knowledge areas. Only in this way will students comprehend the inner content of our field with all its beauty, strength, power, and its nearly unlimited prospects.

Acknowledgments

We thank Cliff Jones for his constructive input.

References

[1] Pierre Bourque and Richard E. Fairley (Eds.). 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK®): Version 3.0 (3rd. ed.)*. IEEE Computer Society Press, Washington, DC, USA.

[2] Manfred Broy, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda, and Daniel Ratiu. 2010. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proc. IEEE* 98, 4 (2010), 526–545. DOI: https://doi.org/10.1109/JPROC.2009.2037771

[3] Achim D. Brucker and Amy Stell. 2023. Verifying feedforward neural networks for classification in vIsabelle/HOL. In *Proceedings of the International Symposium on Formal Methods*. Springer, 427–444. DOI: https://doi.org/10.1007/978-3-031-27481-7_24

[4] Andrew Campbell, Alan Both, and Qian Sun. 2019. Detecting and mapping traffic signs from Google Street View images using deep learning and GIS. *Computers Environment and Urban Systems* 77, 101350 (2019), 1–11. DOI: https://doi.org/10.1016/j.compenvurbsys.2019.101350

[5] Edmund M. Clarke, Jeannette M. Wing et al. 1996. Formal methods: state of the art and future directions. *ACM Computing Surveys* 28, 4 (1996), 626–643. DOI: https://doi.org/10.1145/242223.242257

[6] Byron Cook. 2018. Formal reasoning about the security of Amazon web services. In *Proceedings of the Computer Aided Verification, 30th International Conference*. Springer, 38–47. DOI: https://doi.org/10.1007/978-3-319-96145-3_3

[7] Stijn De Gouw, Frank S. De Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. 2019. Verifying OpenJDK sort method for generic collections. *Journal of Automated Reasoning* 62, 1 (2019), 93–126. DOI: https://doi.org/10.1007/s10817-017-9426-4

[8] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Communications of the ACM* 62, 8 (2019), 62–70. DOI: https://doi.org/10.1145/3338112

[9] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1625–1634. DOI: https://doi.org/10.1109/CVPR.2018.00175

[10] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. 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*. ACM, New York, NY, USA, 555–568. DOI: https://doi.org/10.1145/3037697.3037739

[11] Alessio Ferrari and Maurice H. ter Beek. 2023. Formal methods in railways: A systematic mapping study. *ACM Computing Surveys* 55, 4 (2023), 1–37. DOI: https://doi.org/10.1145/3520480

[12] Anthony C. J. Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P. Mulligan, Gustavo Petri, and Nathan Chong. 2023. A verification methodology for the Arm® confidential computing architecture: From a secure specification to safe implementations. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 88:376–88:405. DOI: https://doi.org/10.1145/3586040

[13] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. 2020. The 2020 expert survey on formal methods. In *Proceedings of the Formal Methods For Industrial Critical Systems*. Springer, 3–69. DOI: https://doi.org/10.1007/978-3-030-58298-2_1

[14] Mario Gleirscher and Diego Marmsoler. 2020. Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. *Empirical Software Engineering* 25, 6 (2020), 4473–4546. DOI: https://doi.org/10.1007/s10664-020-09836-5

[15] Mario Gleirscher, Jaco van de Pol, and Jim Woodcock. 2023. A manifesto for applicable formal methods. *Software and Systems Modeling* 22 (2023), 1737–1749. DOI: https://doi.org/10.1007/s10270-023-01124-2

[16] Samuel Greengard. 2023. AI rewrites coding. *Communications of the ACM* 66, 4 (2023), 12–14. DOI: https://doi.org/10.1145/3583083

[17] John Guttag. 2013. *Introduction to Computation and Programming using Python (Spring 2013 edition)*. MIT Press, Cambridge, Mass.

[18] Charles Antony Richard Hoare. 1981. The emperor's old clothes. *Communications of the ACM* 24, 2 (1981), 75–83. DOI: https://doi.org/10.1145/358549.358561

[19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. 2019. The Marabou framework for verification and analysis of deep neural networks. In *Proceedings of the Computer Aided Verification*. Springer, 443–452. DOI: https://doi.org/10.1007/978-3-030-25540-4_26

[20] Jeff Kramer. 2007. Is abstraction the key to computing? *Communications of the ACM* 50, 4 (2007), 36–42. DOI: https://doi.org/10.1145/1232743.1232745

[21] Bertrand Meyer. 1992. Applying "design by contract". *Computer* 25, 10 (1992), 40–51. DOI: https://doi.org/10.1109/2.161279

[22] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. 2013. Testing or formal verification: DO-178C alternatives and industrial experience. *IEEE Softw* 30, 3 (2013), 50–57. DOI: https://doi.org/10.1109/MS.2013.43

[23] David Lorge Parnas. 2010. Really rethinking "Formal Methods." *Computer* 43, 1 (2010), 28–34. DOI: https://doi.org/10.1109/MC.2010.22

[24] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass.

[25] André Platzer. 2018. *Logical Foundations of Cyber-Physical Systems*. Springer. DOI: https://doi.org/10.1007/978-3-319-63588-0

[26] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the ACM Conference on International Computing Education Research*. ACM, New York, NY, USA, 41–50. DOI: https://doi.org/10.1145/3230977.3230981

[27] Sanjit A. Seshia, Ankush Desai, Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue. 2018. Formal specification for deep neural networks. In *Proceedings of the Automated Technology for Verification and Analysis, 16th International Symposium*. Springer, 20–34. DOI: https://doi.org/10.1007/978-3-030-01090-4_2

[28] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal methods: Practice and experience. *ACM Computing Surveys* 41, 4 (2009), 19:1–36. DOI: https://doi.org/10.1145/1592434.1592436

[29] EN50128: Railway applications - Communication, signalling and processing systems Software for railway control and protection systems (2011).

[30] Nick Rozanski and Eowin Woods. 2005. *Applying Viewpoints and Views to Software Architecture*. Open University White Paper.

[31] ISO26262: Road Vehicles - Functional Safety. International standard (2011).

[32] ISO15408: Common Criteria for Information Technology Security Evaluation. International standard (2009).

[33] K. Rustan and M. Leino. 2017. Accessible software verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97. DOI: 10.1109/MS.2017.4121212

[34] Anthony Hall and Roderick Chapman. 2002. Correctness by construction: Developing a commercial secure system. *IEEE Software* 19, 1 (2002), 18–25. DOI: https://doi.org/10.1109/52.976937

[35] Raphael Barbau and Conrad E. Bock. 2020. Verifying executability of SysML behavior models using satisfiability modulo theory solvers. *NIST Technical Report* 8283 (2020). Retrieved from https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8283.pdf

[36] Jeremy Doerr, Conrad E. Bock, and Raphael Barbau. 2022. Verifying executability of SysML behavior models using Alloy analyzer. *NIST Technical Report* 8388 (2022). DOI: https://doi.org/10.6028/NIST.IR.8388

[37] Colin Snook, Michael Butler, Thai Son Hoang, Asieh Salehi Fathabadi, and Dana Dghaym. 2022. Developing the UML-B modelling tools. *Software Engineering and Formal Methods*. Springer, 13765 (2022), 181–188. DOI: https://doi.org/10.1007/978-3-031-26236-4_16

[38] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Communications of the ACM* 58, 4 (2015), 66–73. DOI: https://doi.org/10.1145/2699417

[39] Brijesh Dongol, Catherine Dubois, Stefan Hallerstede, Eric Hehner, Daniel Jackson, Carol Morgan, Peter Müller, Leila Ribeiro, A. Silva, Graeme Smith, and Eerke de Vink. 2023. On formal methods thinking in computer science education. *Accepted by Form. Asp. Comput.*

[40] Maurice H. ter Beek, Ron Chapman, Rance Cleaveland, Hubert Garavel, R. Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Y. Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and L. Zhang. 2023. Formal methods in industry. *Submitted to Form. Asp. Comput.*

[41] Emil Sekerinski, Marsha Chechik, Joao F. Ferreira, John Hatcliff, Michael Hicks, and K. Lano. 2023. Should we teach formal methods or algorithmic problem solving, design patterns, model-driven engineering, software architecture, software product lines, requirements engineering, and security? In *Preparation.*

[42] Stefan Gruner, Apurva Kumar, and Tom Maibaum. 2015. Towards a body of knowledge in formal methods for the railway domain: identification of settled knowledge. *Communications in Computer and Information Science*. 596 (2015), 87–102. DOI: https://doi.org/10.1007/978-3-319-29510-7_5

[43] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (Eds.). 2007. *The Description Logic Handbook*. Springer.

Received 1 July 2023; revised 23 March 2024; accepted 20 May 2024