How Important are Formal Methods and Formal Logic for Software Engineering Education?

Antti Valmari[1,*], Veikko Halttunen[1]

[1]University of Jyväskylä, PO Box 35, FI-40014 University of Jyväskylä, Finland

Abstract

The debate on teaching formal methods has continued for decades. In this paper, we first examine opinions of software professionals and then review curricula recommendations related to the issue. Next, we discuss examples that illustrate the state of the art, and finally present our observations and opinions. Our main conclusions are: Despite huge progress, formal methods are still too expensive for other than specialized use. They need traditional mathematics-style thinking as support, and thus cannot replace it. It could perhaps be a good idea to shift the emphasis of teaching from more formal aspects of logic (such as proof calculi) to logical tools that make it easier to express properties formally (such as second-order and three-valued logics). Less formal use of some ideas from formal methods (such as representation of loop and class invariants as run-time-checked assertions) might deserve more emphasis than it currently receives.

Keywords

formal and informal reasoning, formal methods, specification, predicate logic, undefined expressions

1. Introduction

In this paper, we discuss the following discrepancy. On the one hand, *many educators* consider logic important for programming and computer science. Errors in software are a very big problem, and so-called formal methods use logic (and/or other mathematical formalisms) to detect and eliminate errors. On the other hand, *software professionals* consider logic only of medium importance and other mathematics of medium or low importance. Formal methods are used very little in practice.

Propositional logic is the science of the truth values "false" and "true" and the connectives "not", "and", "or", "if … then" and "if and only if". Its basics are simple, but as usual in mathematics, various kinds of advanced material is available virtually endlessly. Propositional logic is central to the operating principles of computers and other digital circuits. It also underlies the Boolean data type. However, the latter is not as straightforward as is often thought. For instance, while the "and" of logic is commutative, the "and" of many important programming languages is not. This issue will be discussed in Section 6.

Predicate logic can be divided into first-order logic and higher-order logics. There is a domain of discourse, such as integers or first-in-first-out queues on two letters. Propositional logic is extended with variables whose values range over (the set of) the domain of discourse. There are quantifiers "∀" (for every) and "∃" (there is) which bind variables. There may also be variables that are not bound by quantifiers. The truth value of a formula is a function of the values of the variables of the latter kind. The simplest and most important higher-order logic is second-order logic. It adds variables which range over relations or functions on the domain of discourse. A relation variable with one parameter represents a subset of the domain of discourse.

Although logic is widely considered fundamental to mathematics, most mathematical texts rely extensively on natural language and just a little on formal logic. How many times you have seen a truth table when the topic is not related to logic? According to [1] "despite its fundamental role, logic's place is in the background of what we do, not the forefront. From here on, the beautiful symbols $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$,

NWISEd 2025: Workshop on Co-Creating New Ways of Information Systems Education, September 10–11, 2025, Maribor, Slovenia
*Corresponding author.
ava@jyu.fi (A. Valmari); veikko.halttunen@jyu.fi (V. Halttunen)
https://users.jyu.fi/~ava/ (A. Valmari)
0000-0002-5022-1624 (A. Valmari); 0009-0003-5370-6648 (V. Halttunen)

$\sim$, $\forall$ and $\exists$ are rarely written. But we are aware of their meanings constantly." Regarding formalisms more generally, "abuse of notation"[1] is acceptable and common in mathematics.

Formal logic has been extremely successful in computer science [2]. Basics of logic are often considered essential in computer science and software engineering degrees [3, 4].

The use of formal logic for ensuring the correctness of computer programs was suggested more than 50 years ago (e.g., [5]). It has grown into a large collection of *formal methods* for the specification, verification and automatic construction of software and hardware. In them, the first (and sometimes the only) step is to express desired properties of the system using mathematical formalisms. This is often done by using formal logic, possibly together with set theory, but also other formalisms are currently in use. Verification means checking with mathematical or comparable level of certainty that the properties hold. It may be manual, or partly or fully automated by using such tools as theorem provers and model checkers. The main goal is usually to reduce the number of errors in the end product. Formal methods for specification reduce misconceptions about the end users' needs, and formal methods for the other steps reduce implementation errors.

On the other hand, although errors in software are a very big problem, formal methods have not yet conquered the software world. There is a debate on how much and at what level they should be taught in software education. Some consider them absolutely necessary, e.g., [6]: "We argue that formal methods are and have to be an integral part of every computer science curriculum … software developers not being aware of the various benefits of formal methods cannot be called computer scientists or software engineers". While [7] strongly advocates the use of mathematics and formal methods, it admits that "Evidence supporting the importance of mathematics in software engineering practice is sparse."

The paper [8] positions itself in the middle ground: "This article advocates what we call FM thinking: the application of ideas from Formal Methods applied in informal, lightweight, practical and accessible ways." "finally, how to teach students to be rigorous without being mathematically formal."

The paper introduces three levels of formal methods thinking, intending at least level 1 but not level 3 for all software students. Level 1 seems similar to the formality of mainstream mathematics. Much of computer science is at the same level. Indeed, [8] says "Cormen et al. [2009]" (that is, [9]) "provides an excellent example of how Level 1 FM thinking could be used in a data structures and algorithms course. Many of the algorithms presented in their book are described in terms of invariants without any formal mathematics, yet this provides sufficient clarity and justification to understand why an algorithm works." We checked that its "Appendix: Mathematical Background" lacks logic and its index of mathematical symbols does not contain quantifiers.

Roughly speaking, at level 2 of [8], students can write formal assertions using at least propositional and perhaps also predicate logic, and reason informally but rigorously about them. Level 3 adds to this fully formal proving of properties with automatic tools.

Formal software engineering methods are often compared to mathematical methods in traditional engineering disciplines, e.g., [7]. However, an important difference has often been ignored: it is much more expensive to recall tens of millions of vehicles to fix dangerously faulty air bags[2] than it is to send to tens of millions of mobile phones an update that fixes a serious bug. Because of this difference, there is much less pressure, and thus much less motivation, to make software trustworthy than there is to make typical traditional engineering products trustworthy. Indeed, formal methods have been more successful in hardware than in software.

According to our view, moving from informal to formal assertions is more difficult and makes fully formal verification less reliable than papers advertising formal methods typically suggest. In the following we consider the issue by opening the role of formal logic and formal methods in software engineering and computer science education by using several sources of data.

In Section 2, we will provide an overview of empirical surveys on perceived need for skills in software professions. The mainstream opinion is that formal methods are not important enough to be included in the core topics of curricula. Logic is of mediocre importance, and other mathematics of roughly

[1] https://en.wikipedia.org/wiki/Abuse_of_notation

[2] https://www.nhtsa.gov/vehicle-safety/takata-recall-spotlight

the same or less importance than logic. All but one of these studies are significantly old. Fortunately, ACM / IEEE / AAAI curricula guidelines contain more recent material. They are discussed in Section 3. Advances in data science, artificial intelligence and quantum computation have been taken into account in the most recent of them. Even so, and perhaps surprisingly, its mathematics recommendations are very similar to the surveys in Section 2.

Examples illustrating the state of the art of formal verification in programming will be discussed in Section 4. It will turn out that although fully formal verification of significant pieces of software is possible, it requires so much human effort that it is worthwhile only in special cases. Both formalizing the specification and computer-aided checking of the implementation easily become bottlenecks. Concurrency is a particularly promising area for computer-aided verification. However, the discussion in Section 5 suggests that there the problem of getting the specification right may be even worse. In Section 6, attention is drawn to the fact that although mainstream logic only has two truth values "true" and "false", the use of also a third truth value "undefined" might be recommendable when reasoning about software. Finally, we will present some observations and opinions in Section 7.

2. Importance of Logic and Formal Methods According to Surveys

In this section we discuss results of five surveys on the skills needed in software professions. Unfortunately, four of them are relatively old and do not reflect recent advancements in data science, artificial intelligence and quantum computation. (Section 3 will largely fill this gap.) Despite differences in their methodologies, samples, and the time when the surveys were made, they share the messages relevant for our paper: logic is of medium importance, continuous mathematics and linear algebra are of low importance, and other mathematics is mostly in between them in the minds of software professionals and software students. Furthermore, while informal approaches to requirements gathering and testing are considered very important, formal methods for similar purposes are considered very unimportant.

Until the year 2000 or so, software engineers and computer scientists were taught more or less the same mathematics as traditional engineers, with some discrete mathematics and logic added. Then Timothy Lethbridge published a cover feature article in IEEE Computer presenting results that challenged this practice [10]. From pages 49–50: "Some experts contend that software engineers, like all other engineers, ought to learn about chemistry, physics, and continuous mathematics. While some software engineers would benefit from learning this material, our survey shows that considering these topics to be essential is clearly a mistake. Because of the low importance and high forgetability of continuous mathematics and basic science, universities and colleges should either place less emphasis on these topics or they should teach them in a way that makes them more relevant to software engineering students."

An open call for participation was used and 186 responses were received. "Predicate logic" fared better than most areas of mathematics. Together with "probability and statistics" and "set theory", it was in the middle group of 25 topics, while "differential equations", "differential and integral calculus", "linear algebra and matrices", "combinatorics" and "graph theory" were in the group of the 25 topics considered least important. No mathematics or theoretical computer science topic was in the top group of 25. All 14 mathematics and natural science topics were among the 18 most overtaught (in the sense of being forgotten since education) except "queuing theory", which was not taught much.

Also "formal specification methods" was in the middle group. On the other hand, "requirements gathering and analysis" — an informal approach or alternative to specification — was number 5 in importance. "Testing, verification, and quality assurance" — another area where formal methods promise to be of help — was ranked 12th in importance. These two were considered significantly undertaught.

Kitchenham & al. [11] criticised the setting of, and how the data was collected in [10], and chose themselves a different methodology. Their respondents had graduated in either 1995 or 1998 in computer science or software engineering from four English universities. The questionnaire was somewhat modified from that of [10]. It was sent to 240 graduates. Unfortunately, only 30 replied. Despite the differences in methodology and the risk that the sample was not representative, the results were similar to [10]. The similarity applies to the over-emphasis on mathematical topics in particular. Unlike [10],

"formal specification methods" was found to be the most overtaught topic, and "artificial intelligence" the second most. Also "digital electronics and digital logic" was considered overtaught. "Requirements gathering and analysis" and "testing, verification and quality assurance" were found important, but not considered significantly undertaught.

However, [11] was not ready to drop mathematics from the curricula. It cited [7] saying "Surveys of current practices [8]" (that is, our [10]) "reflect reality; many software engineers have not been taught to use discrete mathematics and logic as effective tools ..." and concluded "If he is correct, ... educationalists need to rethink the ways in which mathematics is taught to software engineering undergraduates."

The sample size of a Finnish survey by Puhakka and Ala-Mutka [12] was 212. Respondents were invited by sending email to the members of the Finnish Information Processing Association. The data was gathered in 2004 mostly copying the questions and topics (with permission) of [10], using 72 topics. The scale was from 1 ("not at all") to 5 ("very much"). The topics were organized into 7 groups. The groups received the following average values for their usefulness in work: software engineering 3.2, computer science core 3.1, other topics (physics, marketing, foreign languages, ...) 2.6, applications of software (World Wide Web, security, artificial intelligence, ...) 2.3, theoretical computer science 1.8, mathematics 1.8 and other information technology (microprocessors, digital signal processing, ...) 1.7.

Of individual topics relevant to our discussion, "logics" got 2.7. It was ranked 28th out of the 72 topics. The rank was higher than that of any other mathematics, theoretical computer science, or other information technology topic. "Requirements Gathering and Analysis" was considered much more useful in work (3.9, rank 5) than "Formal Specification Methods" (1.9, 44), and "Testing, Verification and Validation" (3.7, 8) than "Formal Program Verification" (1.6, 53). This survey also contained "specification and design methods" (3.2, 15) as a software engineering topic. Some other scores and ranks are "Sets, Functions and Relations" (2.2, 34), "Probability and Statistics" (2.0, 42), "Linear Algebra and Matrices" (1.7, 50), "Combinatorics" (1.6, 51), "Graph Theory" (1.6, 52), "Differential and Integral Calculus" (1.5, 59) and "Artificial Intelligence" (1.5, 61).

While [10] reported importance as the average of replies to two questions, one measuring practical usefulness and the other measuring influence on thinking in a general sense, [12] reported each separately. The figures above concern practical usefulness. This difference does not change the big picture, because excluding physics, no topic was more than 0.4 higher and no topic was more than 0.8 lower in influence than in usefulness. All mathematics and theoretical computer science topics had higher or roughly the same influence as usefulness. Almost all other topics had the other way round, with the most important exceptions being physics, chemistry, philosophy, and artificial intelligence.

Puhakka and Ala-Mutka pointed out that for most but not all topics, those who replied at least 3 to how much they learnt it in their formal education gave statistically significantly higher (Mann-Whitney-U-test, $p \leq 0.05$) replies to the usefulness question than others. In the case of theoretical computer science and computer science -related mathematics, but not other mathematics, the absolute changes were mostly non-negligible (although not big). The highest absolute changes were $+1.1$ (parallel and distributed computing), $+0.9$ (software metrics) and $+0.9$ (combinatorics). Unfortunately, [12] does not present much further details.

The survey by Surakka [13] compared the opinions of selected Finnish software developers, professors / lecturers and master's students to each other and the respondents of [10]. Again, the sample sizes were small but the results were quite similar to [10]. Table 1 presents those results that are most relevant for our discussion. The scale is from 1 (Not at all important) to 4 (Very important). The results from [10] were converted to this scale by Surakka. Some of Surakka's topics correspond to a combination of two to four topics in [10].

The students found "discrete mathematics", "logic (in particular, propositional and predicate logic)" and "other areas of theoretical computer science (for example, automata)" less important than the software professionals and professors / lecturers, and the differences were statistically significant ($p < 0.01$). The three Finnish groups found logic programming and constraint logic programming far less important than procedural, object-oriented and concurrent programming, and less important than functional programming. Formal methods were not mentioned in [13].

Table 1
Some results by Surakka 2007 [13] about importance of software education topics

	Software developers	Lethbridge's respondents	Professors and lecturers	Master's students
number of respondents	11	186	19	24
Other areas of theoretical comp. sci. (automata)	3.3	2.3	2.9	2.1
Logic (in particular, propositional and predicate)	2.8	2.3	2.9	1.7
Discrete mathematics	2.6	1.9	3.1	1.7
Mathematics for continuous systems	2.0	1.7	1.7	1.3
Physics	1.6	2.0	1.5	1.1
Logic programming	2.3	-	2.6	1.7
Artificial intelligence and knowledge engineering	1.6	1.8	2.5	1.7
Requirements [Software engineering]	3.6	3.1	3.4	3.3
Test [Software engineering]	3.5	3.0	3.8	3.3

Surakka confirms that the importance of physics and continuous mathematics is "very or quite low", and agrees with the "Because of the low importance and high forgetability" comment in [10] cited above. On the other hand, he argues that "theoretical computer science should be compulsory … some other areas of theoretical computer science might be more important than logic. However, the questionnaires described here were not detailed enough to conclude what kind of theoretical computer science should be taught to CS students."

The survey [14] is much more recent than the above four. It cites [10, 11, 13] and says that there is not any recent study on the mathematics needs of software industry. Its respondents ($n = 628$) graduated in Turkey but were working in 13 countries. The results are similar to earlier surveys (p. 96). Much of mathematics knowledge was forgotten, whereas much software knowledge was acquired while in the industry. The mathematics courses whose material was used most often were "Probability & Statistics" and "(Propositional/Predicate) Logic", but even they were not used more often than "sometimes" on the average.

3. Status of Logic and Formal Methods in Curricula Guidelines

ACM together with other organizations publishes infrequently Curriculum Guidelines for one or another information technology degree topic. The most relevant guidelines to our discussion are Software Engineering 2014 [4] and Computer Science Curricula 2023 [3]. In this section we discuss their recommendations regarding the usefulness of logic and formal methods. Probability and statistics, linear algebra, and calculus have grown in importance because of recent advancements in data science, neural network -based artificial intelligence and, to a lesser extent, quantum computation. Even more obviously this applies to artificial intelligence itself. Computer Science Curricula 2023 is new enough to reflect this.

The most recent Software Engineering Curriculum Guidelines [4] is now ten years old. It uses "lecture hours, abbreviated to *hours*, to quantify instructional time" as a measure of the amount of material (page 25). It specifies 467 lecture hours of "what every SE graduate must know" (p. 7).

Of those hours, 50 are devoted to "Mathematical foundations", which consists of ten topics. "Basic logic (propositional and predicate)" is among the nine topics that were considered "essential" (the other option being "desirable"). It is also among the six with the highest intended cognitive skill level "application". It is emphasized that "Logic and discrete mathematics should be taught in the context of their application to software engineering or computer science problems" (p. 48). The distinction between first-order and higher-order logics is not mentioned.

Formal approaches, formal analysis, etc., are mentioned, but given little emphasis. For comparison, testing has 18 hours and contains 14 topics. On the other hand, the course examples contain one focusing on formal methods.

Computer science is a more theory-oriented sibling of software engineering. The preparation of Computer Science Curricula 2023 [3] was data-driven (p. 49). Early on there was a survey with 865 industry respondents. Views on mathematics were collected from its results. Later on 597 educators replied to a survey focused on mathematical requirements (p. 50).

Computer Science Curricula 2023 specifies 270 "instructional hours" of what "*every* computer science graduate **must** know" (p. 29, p. 30, italics and bold in the original). This is much fewer than the 467 in [4] but, on the other hand, [3] lists 483 hours of "KA Core" topics that are "*recommended* for more in-depth study" (p. 37). They seem to be an intermediate category between obligatory and elective that was introduced because there is much less room in a degree than there is essential material.

Not surprisingly, [3] puts more emphasis on mathematics than [4]. The Knowledge Area "Mathematical and Statistical Foundations" contains 55 hours that are obligatory to everyone. Moreover, there are 145 KA Core hours, some subset of which may be required depending on what other KA Core topics the student studies. For instance, no calculus is declared obligatory for everyone, but some calculus is required by advanced courses in hardware, graphics, etc. Altogether 40 hours of such calculus has been specified. Linear algebra has 5 obligatory and 35 KA Core hours.

The numbers 55 and 145 are significantly bigger than the corresponding numbers 37 and 4 in the ten years older previous guidelines (p. 37). This is because "The application of mathematics has increased in computer science" (p. 52). On the other hand, "mathematics should not be the reason why otherwise well-qualified students are kept away from computer science." The guidelines make suggestions for taking into account "students underprepared in mathematics".

Mathematical and Statistical Foundations consist of five knowledge units: Discrete Mathematics (29 obligatory and 11 KA Core hours), Probability (11+29), Statistics (10+30), Linear Algebra (5+35) and Calculus (0+40). No other distinction is made between obligatory and KA Core than "the KA-core can cover these topics in depth and might include more computing-related applications." Recent advances in data science, artificial intelligence and quantum computation have thus not led to a great number of obligatory linear algebra and calculus hours. Logic is one of the eight topics within Discrete Mathematics, described as "truth tables, connectives (operators), inference rules, formulas, normal forms, simple predicate logic". This implies that only very basics of logics are covered. "Proof techniques (induction, proof by contradiction)", apparently in the informal mainstream mathematics sense, is also one of the eight. Elsewhere in [3] there is some coverage of digital logic circuits at the skill level "apply".

Other than these, logic topics are always either in a minor role or at the lowest skill level "explain" (e.g., pp. 317–387). Formal methods are "Non-Core". The logic programming KA Core contains "First order predicate logic vs higher order logic" (p. 133); this (and its repetition on p. 323) is the only place where this distinction or higher-order logic is mentioned. "First-order" is explicitly mentioned in the mathematics requirements of the Artificial Intelligence and Foundations of Programming Languages knowledge areas.

4. Problems with Applying Formal Methods to Traditional Algorithms

As was mentioned in Section 1, the main goal of formal methods is to help avoid errors in the end product. Formal specification should help avoid ambiguities and inconsistencies, and formal verification should help avoid implementation errors. If the specification is formal enough, there is hope that verification could be partially or fully automated. The more it is automated, the less it is prone to human errors. In this section we point out, using examples, that advantage is gained only if the formal specification is understandable enough. The more difficult to understand the formalization of a property is, the bigger is the risk that it does not say what it was intended to say. Therefore, even in the presence of a correct proof that the system has the formalized property, it may happen that the system fails the intended property.

It may seem unlikely that there could be errors in widely used implementations of well-known age-old algorithms. But there has been such a case [15]. A Java library implementation of binary search failed with very big arrays because of an arithmetic overflow when computing an index. The error had

been "lying in wait for nine years or so" until computer memories grew big enough for it to emerge.

The next examples in this section are about *sorting*. Sorting algorithms and their implementations range from simple and inefficient to highly complicated. We will discuss that although sorting is among the easiest tasks to specify, it is not entirely trivial. Furthermore, it is challenging to verify that a complicated sorting algorithm and its implementation are correct.

It is easy to write a formula that says that the output array must be sorted. Let the array be A and its indices range from 0 to $n-1$. Then the formula $\forall i : 1 \leq i < n \rightarrow A[i-1] \leq A[i]$ will do. Also $\forall i : \forall j : 0 \leq i < j < n \rightarrow A[i] \leq A[j]$ will do. However, this alone does not suffice, because it fails to rule out the following inappropriate algorithm: **for** $i := 1$ **to** $n-1$ **do** $A[i] := A[0]$. This example illustrates that the specification also has to say that the output array consists of the same elements as the input array. This is more difficult than one might have expected. Let the input array be I. For instance, it is not sufficient to say that each value in I is also in A (that is, $\forall i : 0 \leq i < n \rightarrow \exists j : 0 \leq j < n \wedge A[j] = I[i]$) and each value in A is also in I. It does not rule out outputting $[1, 2, 2]$ when the input is $[1, 1, 2]$.

There is a mathematical concept that keeps track of both what elements are present and in how many copies. It is called "multiset" or "bag". If it is available in the specification formalism, then it suffices to say that A represents the same multiset as I. Otherwise, if notation for counting the number of occurrences of a value is available, one can specify that for each $0 \leq i < n$, the value $I[i]$ occurs the same number of times in A as in I. A different idea is to declare the existence of a one-to-one function f from $\{0, \ldots, n-1\}$ to itself such that for every i, $I[i] = A[f(i)]$ (that is, A is a permutation of I). It is straightforward in second-order logic, but not in first-order logic.

Although these two ways of saying "A consists of the same elements as I" are not woefully complicated, it takes some effort to convince oneself that they do say so. On the other hand, many sorting algorithms change the contents of the array only by repeatedly swapping two elements. It is intuitively obvious that immediately after a swap the array consists of the same elements as immediately before the swap. It immediately follows by basic reasoning that after any sequence of swaps the array consists of the original elements. This example shows that a formal specification and proof can be less convincing than an informal argument, because the formal specification (the specification alone, without proof) is harder to understand than the informal argument as a whole.

The situation is not the same with counting sort. It is not easy to see from its code that it satisfies the "same elements" property. Its typical informal correctness proofs are so complicated that the risk that a proof contains an undetected error seems bigger than the risk that the multiset formalization or one-to-one function formalization of "same elements" contains an undetected error. Therefore, a trustworthy fully automatic proof that an algorithm satisfies either formalization of "same elements", might add more to our confidence that the algorithm guarantees "same elements" than an informal proof that uses an informal notion of "same elements".

As such, counting sort is good only for small ranges of keys, but this problem can be solved by using it as a subroutine for another algorithm called radix sort. A semi-automatic proof of a Java version of counting sort and of radix sort was discussed in [16]. Here "semi-automatic" means that the theorem prover did much, but far from all, of the work by itself. The big lines of the proofs came from humans in the form of loop invariants, etc. In a couple of places humans made a change that seems trivial to humans but helped the theorem prover a lot, such as the replacement of `res[c[a[j]]] = a[j]` by `int tmp = a[j]; res[c[tmp]] = tmp`. A significant amount of human guidance was needed also at the detailed level. In the case of counting sort, there were altogether 372 307 proof steps, of which 2 228 were guided by humans. The final proof was fully checked, but not fully found, by the theorem prover.

That the proofs are long is partly due to the nature of formal logic proof systems (they break arguments into tiny steps), and partly due to the need to deal with programming-language level phenomena that are ignored in abstract versions of the algorithms. The distribution of work seems to have been the following: humans encoded the main ideas of an informal proof for the theorem prover, which then filled in more than 99 % of the details automatically and the rest with human help. Two things were gained: the transformations from the abstract algorithms to the implementations were proven correct, and, as a by-product, the correctness of the original informal proofs was confirmed.

Checking that errors are not introduced when implementing an algorithm usually contains a big

amount of intellectually low-level work. Therefore, an automatically checked and almost automatically generated proof of the correctness of the transformation is valuable at least in principle. We leave it to the reader to assess whether it was worth the considerable amount of remaining human effort.

The specification was written in Java Modeling Language (JML). It was too difficult to express the "same elements" property in it, but the theorem prover featured an extension for permutations. However, dealing with permutations in automatic proofs is not easy either. Indeed, it was the cause of about one fifth of the times that the theorem prover was helped by humans at the detailed level.

An important property of counting sort is that it is *stable*, that is, it never changes the relative order of two elements with equal keys. Stability is necessary for the use of counting sort in radix sort. Again, it is a property that is relatively easy to grasp informally but quite complicated to formalize. In [16] it was formalized using an extra array that for each $0 \leq i < n$ tells the original location of $A[i]$.

A state-of-the art example of formal specification and verification is [17]. It discusses a machine-checked correctness proof of a highly optimised highly efficient sorting pogram consisting of more than 900 lines of Java code. Also in this case, JML was used. Verification was very extensive and covered even the absence of arithmetic overflows. A bug causing non-termination was detected (and fixed). The project was participated in by both algorithm engineering experts and program verification experts, and required about 4 person-months. The specification was small and convincing. On the other hand, the main theorem prover did not find all of the proof on its own. It was helped by at least one other verification tool of a different kind, and by giving manually some 2500 lines of information in JML.

Several tricks were used to make the tools succeed. When proving that something is sorted, the formula $\forall i : 0 \leq i < n - 1 \rightarrow A[i] \leq A[i + 1]$ was used. However, when reasoning further from the fact that something is sorted, $\forall i : \forall j : 0 \leq i < n \wedge i \leq j < n \rightarrow A[i] \leq A[j]$ was used. Similarly, when proving that two arrays consist of the same elements, a formula based on multisets was used, but when reasoning further from this, at some places, a formula based on one-to-one functions was used. That these two notions are equivalent was proven elsewhere and given to the main theorem prover as an axiom. One subroutine was proven correct by implementing a radically different, less efficiency-optimised subroutine that provided the same service, proving it correct, and proving that their computations mimic each other.

Let us briefly discuss another type of problem: *reachability*. Finding a route from place A to place B requires solving a reachability problem, and so does avoiding memory leaks (more and more memory becomes reserved without being used). In mathematics, reachability is often defined as the existence of some number n and vertices $v_0, v_1, \ldots, v_n$ such that each (v_{i-1}, v_i) is an edge. Unfortunately, the v_i cannot be treated as predicate logic variables, because their number depends on n. Instead, the v_i are actually a function from natural numbers to the vertices in disguise. Again, such a function can be easily defined in second-order but not in first-order logic. Reachability can be defined in second-order logic also with less machinery (without natural numbers) as $\forall P : \neg P(u) \vee P(v) \vee \exists x : \exists y : P(x) \wedge \mathsf{edge}(x, y) \wedge \neg P(y)$. However, it takes some effort to convince oneself that this does encode reachability.

It is known that reachability cannot be specified in first-order logic without the availability of some strong formalism such as Peano arithmetic or set theory. This causes serious challenges for semi-automatic and automatic verification of many programs. In [18], a workaround was developed and applied to the verification of some small programs dealing with linked lists and garbage collection.

5. A Problem with Applying Formal Methods to Concurrent Systems

A *concurrent system* consists of components that execute in parallel. The emphasis is on problems of co-operation, such as deadlocks, livelocks and loss of synchronization. It is often reasonable to treat also the environment where the system operates as a component. Concurrent systems are a promising application area for formal methods, because they are exceptionally difficult for humans to understand; in many cases a specification that is easy enough to understand can be given; and there are many advanced automatic methods of checking a system against a specification.

Consider the problem of delivering a message from place A to place B when the channel between A and B may occasionally lose messages. A commonly used technique is that always when B receives a message, it sends a special message called *acknowledgement* to A to tell that the original message was not lost. After sending the original message, A waits some time for the acknowledgement. If time runs out without the acknowledgement arriving, A sends the message again, to ensure that B gets it. However, it may be that the original message was received by B but the acknowledgement was lost. Then A re-sends the message unnecessarily, and B may receive it twice. Fortunately, by adding an extra bit to each message in a suitable way, B may be prevented from mis-interpreting the re-send as a new message. The story is a little longer than this, to avoid the possibility that A mis-interprets an acknowledgement as referring to a wrong original message. Even so, the scheme can be made to work. It is known as the *alternating bit protocol* [19].

The specification is intuitively simple: under an assumption presented later, the sequence of delivered messages must be the same as the sequence of original messages. Because the protocol does not look at the contents of the messages (other than the alternating bits), two distinct values for the contents suffice for detecting delivery of a wrong message. The alternating bit protocol is so simple that all possibilities of how it can behave can be investigated by a computer (and even manually).

No protocol can succeed, if from some point of time on the channel loses all messages from A to B or all messages from B to A. Therefore, it is necessary to assume that it does not happen. For the alternating bit protocol, this assumption is also sufficient. This assumption is an example of so-called *fairness assumptions*. That every original message is eventually delivered is a *liveness property*, and that a wrong message is never delivered is a *safety property*. Safety properties do not depend on fairness assumptions, but liveness properties usually do. There is a standard theory of safety, liveness and fairness [20].

In the alternating bit protocol, there is no limit to how many times A re-sends the message, if the acknowledgement never arrives. However, this is not how typical real-life systems are expected to behave. Instead, after some number of futile re-sends they should give up and inform the user that there is no connection. Loss of connection may be temporary. If the connection comes back, depending on what was the last message or acknowledgement that got through before the break, A and B may have a different idea of the value of the alternating bit. This can be solved such that A sends a special "empty" message to B and B sends an acknowledgement for it.

It was pointed out in [21] that the standard theory of fairness assumptions is now in trouble. By losing messages in a suitable manner, the protocol can run forever such that the standard fairness assumption is satisfied but no payload message is ever delivered. The problem remains in a less serious form even if fairness is assumed separately for each of the eight kinds of messages (data of one or the other kind, acknowledgement, and empty, each with 0 or 1 as the alternating bit value).

The problem was discussed in more depth in [22], using mutual exclusion as an example. Among other things, it was illustrated that a basic example picked from the homepage of a widely used model checking tool failed to detect a modification that grossly broke the system. Fairness had been modelled insufficiently. This illustrated that formal methods may fall prey to errors made during formalization.

The lesson of this section is that formal methods for liveness properties are unreliable, because it is difficult to model fairness appropriately and it is difficult to know whether an attempt to do so was successful. (In [21, 22], use of the theory in [23] was suggested as a remedy.)

6. The Problem of Undefined Expressions

It is a basic feature of mainstream logics that every function symbol always yields a value. On the other hand, it is common in mathematics and proramming that a function symbol may fail to yield a value. Division by zero is not defined, an array may be indexed out of bounds, an empty data structure does not have the first element, and a function call may fail to return. In mainstream mathematics this problem is usually solved by informal reasoning whose rules are never made fully explicit. In formal methods there are two main schools, each with variants. In this section we discuss this issue.

One school restricts itself to the two familiar truth values "false" and "true". For instance, $\frac{1}{0} > 0$ may be treated as false. However, then either $\frac{1}{0} \leq 0$ must be treated as true, or the very useful principle saying that $a \leq b$ always means the same as $\neg(a > b)$ must be abandoned. Perhaps the most successful approach [24] in this school says that $\frac{1}{0}$ does have a value in real numbers but we do not know what the value is, so one of $\frac{1}{0} > 0$ and $\frac{1}{0} \leq 0$ is true and the other one is false but we do not know which is which. Instead of $\frac{1}{x} > 0$, we can write either $x \neq 0 \wedge \frac{1}{x} > 0$ or $x = 0 \vee \frac{1}{x} > 0$ depending on what we want the outcome be when $x = 0$. Assume that the legal indices of `A` are from 0 to `n` $- 1$. By default, `i*A[n-i]` is incorrectly treated as returning 0 when `i` $= 0$. This can be solved by explicitly stating and verifying that $0 \leq$ `n` $-$ `i` $<$ `n`. Thus, this approach works, although it has some unexpected consequences. For instance, 0 becomes a root of $\frac{1}{x} = \frac{x}{2} + \frac{1}{2x}$.

The other school introduces a third truth value "undefined". This imports issues that are caused by undefinedness or failure of termination into the logic. For instance, that `i*A[n-i]` is not 0 when `i` $= 0$ is now taken into account by the logic itself. As another example, in many programming languages, the right hand side of "and" is evaluated only if the left hand side returns true. This protects such pieces of code as `while(i<n && A[i]<x){++i;}` from illegally evaluating `A[i]<x` when `i` $=$ `n`. If "&&" behaved like the "$\wedge$" of mainstream logic, then `i<n && A[i]<x` would mean the same as `A[i]<x && i<n`, so `A[i]<x` could be evaluated even if `i` $\geq$ `n`. The behaviour of P && Q, including the possibility of failure because of failure of computing P, or failure of computing Q when P yields true, is precisely modelled by $P \wedge (\neg P \vee Q)$ in this kind of logics. A similar thing holds for "or".

There is some evidence that the notion of the undefined truth value is more natural for software engineers than the way two-valued logics deal with undefined expressions [25]. However, it has been difficult to find a logic that does not occasionally yield unintuitive results. For instance, should $\frac{1}{0} = \frac{1}{0}$ be true, false or undefined? Recently, a solution was suggested and carefully analysed [26] that we believe to match the intuition of mainstream mathematics. A strong sign of it being on a healthy basis is that it is complete in the sense of Gödel's completeness theorem. It also matches the way the teaching tool described in [27, 28, 29] has successfully dealt with undefined expressions for many years now.

There are tricks with which two-valued logics can be made to respect the difference between "&&" and "$\wedge$". For instance, `while(i<n && A[i]<x){++i;}` may be converted to `while(i<n){ if(A[i]>=x){break;} ++i; }`. To avoid the fake root 0, instead of asking the roots of $\frac{1}{x} = \frac{x}{2} + \frac{1}{2x}$ one may ask the roots of $x \neq 0 \wedge \frac{1}{x} = \frac{x}{2} + \frac{1}{2x}$. In these examples, informal workarounds achieve something that three-valued logics facilitate doing in the logic. We believe that doing them in the logic is more natural and less error-prone. On the other hand, three-valued logics are non-standard and not widely taught.

7. Concluding Remarks

In this section we present conclusions and opinions based on the observations in earlier sections, with comments on consequences to teaching. They are mostly in the spirit of "lightweight formal methods" [8]: when correctness is not extremely important, fully formal specifications and proofs are too expensive, but less ambitious approaches to formality may be worth the effort.

Often, deficiencies in software systems have their origin in *capturing of user requirements*. Far too many examples can be found in the student data management system that the authors of this paper must use. For instance, if a student first failed an examination and then tried again with better success, the system did originally not let the teacher enter the passed mark. This was later fixed, but not fully: the teacher still cannot enter a higher mark if the student already has a passed mark.

Problems with capturing user requirements is a pivotal issue. Although requirements elicitation and analysis are at a different level of formality than what has been the focus of this paper, we believe that skills that help at system development at more technical and formal levels also help requirements engineering, by improving the ability to predict consequences of requirements and recognize actual user needs from the incomplete and inconsistent information that representatives of the users give.

The topmost level of much interest to the present paper is *writing a specification*. One problem is that

because formalisms tend to be clumsy, a formal specification may be so far from the original intuition that it is difficult to know whether it says what was intended. While in principle it is possible to express almost everything in hardcore first-order set theory, in practice both mathematicians and computer scientists use a lot of second-order machinery. This raises the question whether instead of teaching a complete formal proof system for first-order logic, time would be better spent in teaching how to express ideas in second-order logic. Another possibility for teaching formal tools that are closer to intuition is three-valued logics that can handle undefined expressions.

It is clear from previous sections that with the current state of the art, *formal verification of the correctness* of non-trivial systems requires significant amounts of human work. Furthermore, a formally fully checked proof does not necessarily guarantee correctness as end users see it. "Engineering software always includes a step from the informal to the formal" [6]. Errors may be made when expressing requirements formally, as was discussed above. The original intuition may have been incorrect to start with. It is easy to fail to recognize the need of some requirement, such as that a sorted array must consist of the same elements as the input array. This reduces motivation for paying the often high cost of full formality.

The big lines of a correctness proof of a non-trivial system must come from humans. Verification tools may need human help also with details. The role of the tools is to take care of most details and carefully check the resulting proof. Therefore, to formally verify a system, there must first be a sketch of a rigorous less formal proof of the kind of typical proofs in mathematics. This suggests that teaching hardcore formal verification needs, and thus cannot replace, teaching typical mathematical thinking. The latter is useful also in its own right.

Even so, computer-aided formal verification could in principle be very useful. It is easy for humans to make and difficult to detect superficial errors, such as writing "&" instead of "&&". As a consequence, an implementation may be incorrect although the implemented algorithm is correct. Unfortunately, problems with writing specifications and insufficient performance of verification tools are big obstacles.

A less ambitious idea is to add to the program pieces of code that check various assertions and report their violations, and run the program a large number of times with diverse input data. If any loop or class invariants are known that would be used in a formal proof, they could be represented as such pieces of code. For a programmer, expressing an invariant as program code is likely to be easier than expressing it as a formula. The diverse input data could be just random data or designed to test the program as extensively as possible. Because computers are fast, big amounts of random data may be used. This approach might deserve more attention in curricula than it currently gets.

Regarding teaching of logic, [30] identified three issues. One of them was the topic of Section 6.

Another issue is that the formal concepts of logical consequence, logical equivalence and tautology are not the similar concepts that mainstream mathematicians and computer scientists need. This is because in mainstream mathematics, familiar symbols such as $+$, 1 and $\leq$ have their familiar meanings, while logical consequence, etc., recognize only what has been explicitly written in the formulas. This difference is subtle but far-reaching, and has caused lots of misunderstandings. As a consequence, teaching logical consequence, etc., should be replaced by teaching mathematical consequence, etc.

Finally, the notion of implication seems particularly difficult for humans. For instance, [31] says: "Despite the overwhelming presence of such sentences in everyday discourse and reasoning, there is surprisingly little agreement about what the right logic of conditionals might be, or even about whether a unified theory can be given for all kinds of conditionals." The paper [30] made many suggestions regarding how implication should be taught. One of them is to clearly distinguish between implication as a logical connective and as a reasoning operator.

New kind of tool support for teaching logic is discussed in [32].

Altogether, it seems that hardcore formal methods are not yet practical enough to justify other than elective role in software curricula. Advanced formal aspects of logic (such as proof calculi) do not deserve a strong role either. Effort may be better spent teaching how to write rigorous definitions and proofs, where "rigorous" refers to the mixture of formal and informal of the kind that is usual in mainstream mathematics and traditional core computer science. Even if a degree program has particular interest in formal logic or formal methods, the road to them goes via rigorous less formal thinking.

Acknowledgments

The authors thank the reviewers for their efforts and feedback.

Declaration on Generative AI

The authors have not employed any Generative AI tools except for minor English language improvement.

References

[1] R. Hammack, Book Of Proof, 3rd ed., Self-published, 2018. URL: https://richardhammack.github.io/BookOfProof/, approved by the American Institute of Mathematics' Open Textbook Initiative.

[2] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, V. Vianu, On the unusual effectiveness of logic in computer science, Bulletin of Symbolic Logic 7 (2001) 213–236. doi:10.2307/2687775.

[3] The Joint Task Force on Computer Science Curricula: Association for Computing Machinery (ACM), IEEE-Computer Society (IEEE-CS), Association for the Advancement of Artificial Intelligence (AAAI), Computer Science Curricula 2023, ACM Press, IEEE Computer Society Press and AAAI Press, 2024. doi:10.1145/3664191.

[4] Joint Task Force on Computing Curricula: IEEE Computer Society, Association for Computing Machinery, Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, ACM, New York, NY, USA, 2015. URL: https://www.acm.org/binaries/content/assets/education/se2014.pdf.

[5] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580. doi:10.1145/363235.363259.

[6] M. Broy, A. D. Brucker, A. Fantechi, M. Gleirscher, K. Havelund, M. A. Kuppe, A. Mendes, A. Platzer, J. O. Ringert, A. Sullivan, Does every computer scientist need to know formal methods?, Formal Aspects Comput. 37 (2025) 6:1–6:17. doi:10.1145/3670795.

[7] P. B. Henderson, Mathematical reasoning in software engineering education, Commun. ACM 46 (2003) 45–50. doi:10.1145/903893.903919.

[8] B. Dongol, C. Dubois, S. Hallerstede, E. C. R. Hehner, C. Morgan, P. Müller, L. Ribeiro, A. Silva, G. Smith, E. P. de Vink, On formal methods thinking in computer science education, Formal Aspects Comput. 37 (2025) 8:1–8:23. doi:10.1145/3670419.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[10] T. Lethbridge, What knowledge is important to a software professional?, IEEE Computer 33 (2000) 44–50. doi:10.1109/2.841783.

[11] B. A. Kitchenham, D. Budgen, P. Brereton, P. Woodall, An investigation of software engineering curricula, Journal of Systems and Software 74 (2005) 325–335. doi:10.1016/j.jss.2004.03.016.

[12] A. Puhakka, K. Ala-Mutka, Survey on the Knowledge and Education Needs of Finnish Software Professionals, Technical Report, Tampere University of Technology, 2009. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f0a4e83dab64e3fdc9310ac905435ddf19bc61a5.

[13] S. Surakka, What subjects and skills are important for software developers?, Communications of the ACM 50 (2007) 73–78. doi:10.1145/1188913.1188920.

[14] D. Akdur, The analysis of mathematical skills used in the software industry, Turkish Journal of Mathematics and Computer Science 12 (2020) 92–100. doi:10.47000/tjmcs.789945.

[15] J. Bloch, Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken, 2006. URL: https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html, [Online; accessed 2025-05-27].

[16] S. de Gouw, F. S. de Boer, J. Rot, Proof pearl: The KeY to correct and stable sorting, J. Autom. Reason. 53 (2014) 129–139. doi:10.1007/S10817-013-9300-Y.

[17] B. Beckert, P. Sanders, M. Ulbrich, J. Wiesler, S. Witt, Formally verifying an efficient sorter, in: B. Finkbeiner, L. Kovács (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I, volume 14570 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 268–287. doi:10.1007/978-3-031-57246-3_15.

[18] T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, G. Yorsh, Simulating reachability using first-order logic with applications to verification of linked data structures, Log. Methods Comput. Sci. 5 (2009). URL: http://arxiv.org/abs/0904.4902.

[19] K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson, A note on reliable full-duplex transmission over half-duplex links, Commun. ACM 12 (1969) 260–261. doi:10.1145/362946.362970.

[20] Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems - specification, Springer, 1992.

[21] A. Valmari, W. Vogler, Fair testing and stubborn sets, Int. J. Softw. Tools Technol. Transf. 20 (2018) 589–610. doi:10.1007/S10009-017-0481-2.

[22] A. Valmari, H. Hansen, Progress checking for dummies, in: F. Howar, J. Barnat (Eds.), Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings, volume 11119 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 115–130. doi:10.1007/978-3-030-00244-2_8.

[23] A. Rensink, W. Vogler, Fair testing, Inf. Comput. 205 (2007) 125–198. doi:10.1016/J.IC.2006.06.002.

[24] D. Gries, F. B. Schneider, Avoiding the undefined by underspecification, in: J. van Leeuwen (Ed.), Computer Science Today: Recent Trends and Developments, volume 1000 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 366–373. doi:10.1007/BFB0015254.

[25] P. Chalin, Logical foundations of program assertions: What do practitioners want?, in: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany, IEEE Computer Society, 2005, pp. 383–393. doi:10.1109/SEFM.2005.26.

[26] A. Valmari, L. Hella, A completeness proof for a regular predicate logic with undefined truth value, Notre Dame Journal of Formal Logic 64 (2023) 61–93. doi:10.1215/00294527-2022-0034.

[27] T. Kaarakka, K. Helkala, A. Valmari, M. Joutsenlahti, Pedagogical experiments with MathCheck in university teaching, LUMAT: International Journal on Math, Science and Technology Education 7 (2019) 84–112. doi:10.31129/LUMAT.7.3.428.

[28] A. Valmari, J. Rantala, Arithmetic, logic, syntax and MathCheck, in: H. Lane, S. Zvacek, J. Uhomoibhi (Eds.), Proceedings of the 11th International Conference on Computer Supported Education, CSEDU 2019, Heraklion, Crete, Greece, May 2-4, 2019, Volume 2, SciTePress, 2019, pp. 292–299. doi:10.5220/0007708902920299.

[29] A. Valmari, Automated checking of flexible mathematical reasoning in the case of systems of (in)equations and the absolute value operator, in: Proceedings of the 13th International Conference on Computer Supported Education, CSEDU 2021, Online Streaming, April 23-25, 2021, Volume 2, SCITEPRESS, 2021, pp. 324–331. doi:10.5220/0010493103240331.

[30] A. Valmari, Adapting formal logic for everyday mathematics, in: M. Cukurova, N. Rummel, D. Gillet, B. M. McLaren, J. Uhomoibhi (Eds.), Proceedings of the 14th International Conference on Computer Supported Education, CSEDU 2022, Online Streaming, April 22-24, 2022, Volume 2, SCITEPRESS, 2022, pp. 515–524. doi:10.5220/0011063300003182.

[31] P. Egré, H. Rott, The Logic of Conditionals, in: E. N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy, Winter 2021 ed., Metaphysics Research Lab, Stanford University, 2021. URL: https://plato.stanford.edu/archives/win2021/entries/logic-conditionals/.

[32] A. Valmari, Automatic feedback on logic problems about real numbers and integers, in: 3rd Workshop on Co-Creating New Ways of Information Systems Education, Sept. 2025, Maribor, Slovenia, CEUR Workshop Proceedings, CEUR-WS.org, 2025. Accepted to.