Latest updates: https://dl.acm.org/doi/10.1145/3670419

PDF Download
3670419.pdf
26 February 2026
Total Citations: 12
Total Downloads:
2888

RESEARCH-ARTICLE

On Formal Methods Thinking in Computer Science Education

Published: 26 December 2024
Online AM: 01 June 2024
Accepted: 20 May 2024
Revised: 27 March 2024
Received: 24 July 2023

Citation in BibTeX format

BRIJESH DONGOL, University of Surrey, Guildford, Surrey, U.K.

CATHERINE DUBOIS, Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise, Evry, France

STEFAN HALLERSTEDE, Aarhus University, Aarhus, Midtjylland, Denmark

ERIC HEHNER, University of Toronto, Toronto, ON, Canada

CARROLL MORGAN, UNSW Sydney, Sydney, NSW, Australia

PETER MÜLLER, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

View all

Open Access Support provided by:

University of Toronto

Cornell University

Eindhoven University of Technology

The University of Queensland

University of Surrey

Federal University of Rio Grande do Sul

View all

On Formal Methods Thinking in Computer Science Education

BRIJESH DONGOL, University of Surrey, Guildford, United Kingdom of Great Britain and Northern Ireland
CATHERINE DUBOIS, Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Enterprise, Evry, France
STEFAN HALLERSTEDE, Aarhus Universitet, Aarhus, Denmark
ERIC HEHNER, University of Toronto, Toronto, Canada
CARROLL MORGAN, University of New South Wales, Sydney, Australia
PETER MÜLLER, ETH Zurich, Zurich, Switzerland
LEILA RIBEIRO, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
ALEXANDRA SILVA, Cornell University, Ithaca, United States
GRAEME SMITH, School of EECS, The University of Queensland, Brisbane, Australia
ERIK DE VINK, Eindhoven University of Technology, Eindhoven, Netherlands

Formal Methods (FMs) radically improve the quality of the code artefacts they help to produce. They are simple, probably accessible to first-year undergraduate students and certainly to second-year students and beyond. Nevertheless, in many cases, they are not part of a general recommendation for course curricula, i.e., they are not taught — and yet they are valuable.

One reason for this is that teaching "Formal Methods" is often confused with teaching logic and theory. This article advocates what we call *FM thinking*: the application of ideas from *Formal* Methods applied in informal, lightweight, practical and accessible ways. We will argue here that *FM thinking* should be part of the recommended curriculum for every Computer Science student, for even students who train only in that "thinking" will become much better programmers. However, there will be others who, exposed to those ideas, will be ideally positioned to go further into the more theoretical background: why the techniques work, how they can be automated, and how new ones can be developed. Those students would follow subsequently a specialised, more theoretical stream, including topics such as semantics, logics, verification and proof-automation techniques.

Authors' Contact Information: Brijesh Dongol, University of Surrey, Guildford, United Kingdom of Great Britain and Northern Ireland; e-mail: b.dongol@surrey.ac.uk; Catherine Dubois, Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Enterprise, Evry, Île-de-France, France; e-mail: catherine.dubois@ensiie.fr; Stefan Hallerstede, Aarhus Universitet, Aarhus, Midtjylland, Denmark; e-mail: sha@ece.au.dk; Eric Hehner, University of Toronto, Toronto, Ontario, Canada; e-mail: hehner@cs.utoronto.ca; Carroll Morgan, University of New South Wales, Sydney, Australia; e-mail: carroll.morgan@unsw.edu.au; Peter Müller, ETH Zurich, Zurich, Switzerland; e-mail: peter.mueller@inf.ethz.ch; Leila Ribeiro, Federal University of Rio Grande do Sul, Porto Alegre, RS, Brazil; e-mail: leila@inf.ufrgs.br; Alexandra Silva, Cornell University, Ithaca, New York, United States; e-mail: alexandra.silva@cornell.edu; Graeme Smith, School of EECS, The University of Queensland, Brisbane, Queensland, Australia; e-mail: smith@itee.uq.edu.au; Erik de Vink, Eindhoven University of Technology, Eindhoven, Noord-Brabant, Netherlands; e-mail: evink@win.tue.nl.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1433-299X/2024/12-ART8
https://doi.org/10.1145/3670419

CCS Concepts: • **Theory of computation → Program reasoning**; • **Applied computing → Education**; • **Software and its engineering**;

Additional Key Words and Phrases: Formal methods, ACM CS2023, computer-science education

ACM Reference Format:
Brijesh Dongol, Catherine Dubois, Stefan Hallerstede, Eric Hehner, Carroll Morgan, Peter Müller, Leila Ribeiro, Alexandra Silva, Graeme Smith, and Erik de Vink. 2024. On Formal Methods Thinking in Computer Science Education. *Form. Asp. Comput.* 37, 1, Article 8 (December 2024), 23 pages. https://doi.org/10.1145/3670419

1 Introduction

Professional engineering maintains its high reputation in our society by insisting that, to be a professional engineer, one must know and apply the relevant theories. A civil engineer must know and apply the theories of geometry and material stress. An electrical engineer must know and apply electromagnetic theory. Software engineers, to be worthy of the name, must know and apply the theory of programming and software development, which includes **Formal Methods (FM).** But what do we mean when we refer to FM in **Computer Science (CS)**? This area encompasses approaches and techniques (and tools) to aid the construction of computational systems (including software and hardware) by providing rigorous means for modelling and analysis. It provides precise and unambiguous explanations for the behaviour of programs and for concepts such as modularisation, refinement, inheritance, and model transformations. That is why FM is a "relevant theory" of *software* engineering in the same sense as the other engineering theories mentioned above.

One may ask *What good is FM? Who needs it? Millions of programmers work every day without it.* Many think that FM in a CS curriculum is peddling the idea that Formal Logic (e.g., propositional or predicate logic) is required for everyday programmers, that they need it to write programs that are more likely to be correct, and correspondingly less likely to fail the tests to which they subsequently (of course) must still be subjected. However, this degree of formality is not necessarily needed. What is required of everyday programmers is that, as they write their programs, they think — and code —in a way that *respects a correctness-oriented point of view.* Assertions can be written informally, in natural language: just the "thinking of what those assertions might be" guides the program-construction process in an astonishingly effective way. What is also required are the engineering principles referred to above. Connecting programs with their specifications through assertions provides training on *abstraction*, which, in turn, encourages simplicity and focus, helping build more robust, flexible and usable systems.

The answer to "Who needs it?" is that everyday programmers and software developers indeed may not need to know the *theory* of FM. But they do need to know how to *practise* it, even if with a light touch, benefiting from its precepts. FM *theory*, which is what explains — to the more mathematically inclined — why FM works, has become confused with the FM *practice* of using the theory's results to benefit from what it assures. Any "everyday programmer" can do that…except that most do not. Why not? Because we do not teach them how.[1] We must move away from the idea that the CS curriculum should only include advanced courses, the ones that plumb the theory,

[1]To catch a fly ball, one should wait in a spot where its upwards speed seems to be constant: that is baseball *practice*. To know why that works, one needs trigonometry and high-school physics: that is baseball *theory*. The player in the outfield knows where to stand (practice), but does not need to know why it works (theory). Even the player's coach does not need to know the theory: it is the practice that must be passed on.

that create the engineers who can prove correctness of whole operating systems. We must also develop courses that train educators so that they, in turn, find, nurture and encourage students to use FM thinking: we must teach the teachers!

In summary, much mundane programming can be done without direct appeal to a theory. However, the more difficult programming is very unlikely to be done correctly without a good theory that "behind the scenes" validates the methods being used. The software industry has an overwhelming experience of buggy programs to support that statement. And even mundane programming can be improved by the use of a theory. Our ability to control and predict motion changes from an art to a science when we learn mathematical theory. Programming changes from an art to a science in a similar way.

The reality is that real-world software is unreliable, and most of it is developed without rigorous arguments about correctness. Indeed, much of it does not need rigorous arguments: can you be rigorous about "ease of use"? Probably not, but that does not exclude being rigorous where it is necessary.

Contributions: FM versus FM thinking. FM is at the heart of CS, providing mechanisms for understanding why programs (and computations in general) are correct. Many FM techniques have been developed, ranging from lightweight methods[2] (e.g., type checking, static analysis, runtime verification, fuzzers and linters) to heavyweight methods (e.g., full proofs of pre- and post-conditions, model checking and theorem proving). While lightweight methods are now being regularly used in industry [Garavel et al. 2020], heavyweight methods (which provide stronger correctness guarantees) are still seen as being too difficult and costly to use in an industrial setting. This is reflected in our teaching, in which many CS curricula simply bypass this material in the first and second year and reserve them for advanced courses.

This article proposes an alternative approach and addresses the fundamental question in CS education of how to achieve the goals discussed above: how to teach to be rigorous when rigour is demanded (e.g., in safety-critical systems, cloud services, etc.); how to teach to recognise when rigour is not applicable, maybe even an impediment; and, finally, how to teach students to be rigorous without being mathematically formal.

We advocate a need for FM thinking for all CS students, i.e., the use of basic FM principles to describe the correctness of a system even when used at the most basic level without any formal mathematical theory. We propose gradations of FM education covering *lightweight FM thinking* (which allows students to explain why the programs they have written are correct at the most basic level); *intermediate FM thinking* (which focuses on a higher level of precision, on assertions in propositional logic, probabilistic and arithmetic statements), and *heavyweight FM thinking* (which focuses on full proofs of correctness of a safety or security property, often with tools such as model checkers and theorem provers, and with respect to a comprehensive formal specification).[3]

Achieving our goals requires changes to the way CS is taught (even in the first year of university). These changes can be introduced without creating additional bloat. However, if these changes are to be implemented fully, we must also train educators, i.e., teachers who are sufficiently proficient in "advanced" (i.e., intermediate/heavyweight) FM and, hence, are capable of motivating FM thinking in the CS curriculum.

[2]A slightly different notion of "lightweight FM" is used by Jackson and Wing [1996], in which partiality (e.g., of specifications, languages, modelling and analysis) is encouraged to reduce the cost of deploying FM and increasing tractability.
[3]The terms "lightweight" and "heavyweight" when applied to FM thinking should not be confused with lightweight and heavyweight FM techniques [Garavel et al. 2020], although our intention is for heavyweight FM thinking to support both lightweight and heavyweight FM techniques.

Article structure. We present our notion of "FM thinking" in Section 2 and introduce three levels of its mastery: from informal natural-language reasoning to tool-based verification and validation. In Section 3, we discuss how these can be embedded in a standard CS curriculum without displacing other material. In Section 4, we reflect on the *ACM 2023 Knowledge Areas* [ACM 2023; Kumar et al. 2023]. As with the courses in Section 3, we discuss how levels of FM thinking can be introduced into these knowledge areas without increasing the suggested number of hours that the material demands. Finally, in Section 5, we describe pathways to teaching CS educators to introduce more FM thinking into their courses.

2 Formal Methods Thinking

In this section, we motivate the benefits of FM thinking (Section 2.1). In Section 2.2, we provide further details on the three levels of FM thinking. In Sections 2.3 and 2.4, we provide examples on how FM thinking can be applied at these different levels.

2.1 How CS Students Benefit from FM Thinking

For many students, the first understanding of the programs that they are taught is based on how these programs are executed. For many students, that is all they are given. With that understanding, the only method available for checking whether a program is correct is to test it by executing it with a variety of inputs to see whether the resulting outputs are right. All programs should be tested, using FM or otherwise. However, there are three problems with testing without FM thinking.

First problem How does one know whether the outputs are right? For some programs, such as graphics programs for producing pleasing pictures, the only way to know whether the output is right is to test the program and judge the result, probably subjectively. In other cases, a program might give answers students do not already know (which could well be why they wrote the program) — and then testing it does not tell them whether it is right. In such cases, the fallback is that students should test to see at least whether the answers are reasonable.

Second problem You cannot try all inputs. Even if all the test cases you try give reasonable answers, there could still be errors lurking in untried cases.

Third problem The purpose of testing is not to guarantee that the program is correct but rather to find bugs. Conceptually, these are very different tasks that require distinct ways to reason about a program.

A course on FM gives students an understanding of programs that is completely independent of individual execution cases. This is achieved by articulating the notion of a *specification* that covers the expected behaviour in all cases, even if these are stated informally. Students should learn that a specification should capture *what* the program achieves, rather than *how* it achieves its end goal.

When one proves that a program satisfies a specification, one is considering all inputs at once and proving that the outputs have the properties stated in the specification. That is far more than can ever be accomplished by testing alone. Note that we do not advocate replacing testing with FM. In fact, one of the reasons that testing remains important is that it is possible to make mistakes even in the formal reasoning — not only mistakes in logical calculations (which tools help to avoid) but also mistakes in the specification itself.

What FM does is to augment testing with techniques that make mistakes (of any kind) much less likely. The underlying learning objective is that programs seek to establish or maintain properties and that those properties need to be expressed and can be reasoned about. FM helps us to write precise specifications and to design programs whose executions provably satisfy the specifications. FM improves our argumentation skills and provides the basis on which solid reasoning about software systems' behaviour can be built. Moreover, FM instils a mindset of reflecting on our designs and checking (or verifying) that the intentions (or requirements) are met.

A natural question to ask here is, then, "How much of this rigour actually requires mathematics?" The answer is that FM *thinking* can be informal in the way it is applied [Morgan 2014]. Even if it includes formality, it is most often an application of principles from a subset of discrete mathematics — predicate logic and basic set theory — which is already part of the existing CS curriculum. Only for those students interested in pursuing more theoretical electives will additional mathematics be required, e.g., in some domains (cyberphysical systems, robotics, security, networks, etc.) students will be required to go beyond discrete mathematics to be able to express and articulate the probabilistic and continuous properties of the world around them.

2.2 Levels of FM Thinking

As an analogy, most people do not like doing arithmetic in their heads. However, if they are working in a restaurant and having to add up dinner bills (think some decades ago), they will be happy to learn to do it right, on paper (first stage), because otherwise they will pay for their mistakes. So then (second stage), they will thank you for a reliable method even if still by hand, and (third stage) ultimately they will be glad to have a calculator.

2.2.1 Level 1 ("What's True Here"). Level 1 of FM thinking is the application of FM in its most basic form. Students develop abilities to understand their programs and reason about their correctness using informal descriptions. By "What's True Here", we mean including natural language prose or informal diagrams[4] to describe the *properties that are true* at different points of a program's execution rather than the operations that brought them about.

The student competencies (and their associated Bloom's Taxonomy[5] levels [Anderson et al. 2001; Bloom et al. 1956]) that are expected to be gained from this level of FM thinking include:

(Apply level) Formulate assertions that hold at given points in a program using simple rules.
(Analyse level) Explain the behaviour of a program in terms of such assertions.
(Evaluate level) Justify the correctness of a program in terms of such assertions. Argue why a program satisfies the given requirements.

Level 1 can be incorporated into any undergraduate program and could often come even before any code is written. Before we show our students how to build their first data structure, we must first explain what effects the operations on the data structure have: as the operation executes we must be able to describe what we are expecting each step to have brought about. Cormen et al. [2009] 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. Other examples include graphical notations for teaching programming and loop invariants, which are accompanied by both informal [Brieven et al. 2023] and formal [Back 2009] semantics.

Morgan [2014] gives an extensive description, almost a "teacher's manual", of how the so-called "What's True Here?" thinking and documenting can be introduced even to early-curriculum CS students, moving them gently away –even at that stage– from the less effective (but currently

[4]Of course, diagrammatic notations such as message sequence charts and state transition diagrams can have a formal semantics. Back [2009] has developed a formal diagrammatic notation for teaching programming from invariants.
[5]Bloom's Taxonomy is a widely used framework for setting educational objectives, describing learning across cognitive domain levels: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation. The taxonomy has been revised several times since its inception. We use the formulation by Anderson et al. [2001], who define Bloom's Taxonomy as comprising these levels: Remember, Understand, Apply, Analyse, Evaluate, and Create.

widespread) use of the "What This Code Did" -style comments that they would otherwise acquire in an ad-hoc manner.

Another example is Liskov and Guttag's book [Liskov and Guttag 2001], *Program Development in Java — Abstraction, Specification and Object-Oriented Design*. This is essentially an introductory text on Java that at the same time introduces concepts such as procedural abstraction using pre- and postconditions (written in natural language) and data abstraction using abstraction functions and representation or class invariants (again, written in natural language). The book explains how, for example, abstraction functions are the basis for information hiding and encapsulation, and pre- and postconditions the basis for subtyping: both are necessary to write good programs.

A further relevant book is *How to Design Programs: An Introduction to Programming and Computing* [Felleisen et al. 2018], which emphasises best software engineering practices for the Racket language, like the use of informal arguments to justify termination of recursive functions. To understand the idea of recursion, which is central in CS, one does not need a great theoretical background. Once this concept is understood, it becomes easy to explain and reason about the behaviour of a plethora of programs.

Ultimately, as the systems become more complicated, so do the notions of "What's True Here?" and their manipulations. It becomes difficult to remember all the natural-language assertions, and inconsistencies begin to emerge — perhaps the same phrase has been used to describe different phenomena in two different places or perhaps it is not really clear that one statement implies another. At that point, students are ready for the next two levels: firstly, writing down the assertions in a more formal manner so that ambiguity is diminished or even avoided altogether (Level 2); and secondly, when even that becomes too much to handle, using a tool such as Dafny [Leino 2010] to help keep track of and reason about their programs (Level 3).

2.2.2 Level 2 (Formal Assertions). Level 2 introduces greater precision to Level 1 by teaching students to write assertions that incorporate arithmetic and logical operators to capture FM thinking more rigorously. This may be accompanied by lightweight tools that can be used to test or check that their assertions hold.

The student competencies expected to be gained from this level of FM thinking include:

(Apply level) Using rules and *laws of propositional or predicate logic* to derive *mathematically precise* assertions that hold at given points in a program.

(Analyse level) Explaining the behaviour of a program in terms of such assertions, potentially with the *assistance of runtime testing tools* and *runtime assertion checkers*.

(Evaluate level) Justifying the correctness of a program in terms of such assertions and a *mathematically precise argument*.

All programmers know propositional logic and arithmetic operators from having to write Boolean expressions in their code (e.g., within conditionals and loops). What they might not be familiar with is writing Boolean expressions *about* their code. Within Level 2, one can consider different gradations on the complexity of the language of assertions, and it is important to understand the limitations on the properties that can ultimately be captured. Access only to propositional logic means that programs with arrays (and other collection types) cannot be considered (as those will inevitably require conditions with quantifiers). However, most students will have encountered predicate logic in their first-year discrete-mathematics courses.

Depending on the program, students may also have to be taught to write assertions that go beyond propositional and predicate logic. Potentially, they must also include formal descriptions using probabilities and continuous functions, e.g., to describe assumed properties about a program's real-time environment. Writing down properties that specify how often a variable is expected to change would justify the rate at which the variable is polled by the program. Another program might only be able to provide a probabilistic guarantee to its user because the environment in which the program is used is itself probabilistic.

Ultimately, proficiency at Level 2 is needed to ensure that students develop assertions with greater precision and rigour than at Level 1. However, there are still several reasons why students might choose to go beyond that and progress to Level 3:

— The formulas become increasingly complex as the complexity of the program being verified increases, making it more difficult to understand or remember why a particular property follows from another.
— The problem domain demands that properties be mechanically checked.
— The tools are easy enough to use so that the properties that they are checking can be automatically validated.

Once a student has learned to reason about programs with assertions, the verification tools act as a calculator that can be used to check that a student's understanding of the correctness of their programs is accurate.

2.2.3 Level 3 (Full Verification). This level enables students to prove program properties using tools such as a theorem prover, model checker or SMT solver. But in addition to tool-based checking of properties (now written using a formal language), this level can formally emphasise other aspects of system-level correctness, such as structural induction and termination.[6]

Student competencies expected to be gained from this level of FM thinking include:

(Apply level) Using simple rules and laws of *predicate logic* to derive mathematically precise assertions that hold at given points in a program.
(Analyse level) Explaining the behaviour of a program in terms of such assertions and with the assistance of *design-time analysis tools*.
(Evaluate level) Justifying the correctness of a program in terms of such assertions and a mathematically precise and *machine-checked* argument.

FM thinking at Level 3 can cover different types of program semantics (operational, axiomatic, denotational) to give a rigorous definition to the meaning of program execution. This can be accompanied by classical topics such as Hoare logic and weakest preconditions as well as modern variations, including separation logic. There can also be focus on system-level reasoning using different temporal logics such as **Linear Temporal Logic (LTL)** and **Computation Tree Logic (CTL)**, and explicit-state model checking. The FM itself becomes a platform to explore topics in concurrency, where natural language reasoning seldom works. Moreover, notions such as abstraction and refinement and related proof techniques such as (bi)simulation, which are covered to some degree in other CS courses (e.g., software engineering), can be defined and used both to reason about and to develop systems.

Level 3 can ask students to master a wide variety of tools. For example, the Rigorous Software Engineering course at ETH uses Alloy [Jackson 2012, 2019] to teach modelling and software design, and introduces abstract interpretation (building on the operational semantics). Those topics lead to further pathways in the masters-level curriculum, e.g., in security, programming languages, and machine learning. A more comprehensive list of Level 3 courses and the tools they use can be found at https://fme-teaching.github.io/courses/. Popular tools include model checkers (NuSMV, Spin, Prism, Uppaal, CADP, mCRL2), static analysers (Frama-C), SMT solvers (Z3, CVC5),

[6]Note that concepts such as structural induction may appear earlier in a curriculum alongside a functional programming course without formal proofs [Felleisen et al. 2018; Gibbons 2021]. (Similarly, termination).

```
t := x;
x := y;
y := t;
```

Fig. 1. Swapping variables using an extra temporary variable.

```
x,y := y,x;
```

Fig. 2. Swapping variables using parallel assignment.

```
x := x + y;
y := x - y;
x := x - y;
```

Fig. 3. Swapping variables using only addition and subtraction.

deductive verifiers (Why3, Dafny, Whiley, Key, Logika), formal modellers (Rodin, Atelier-B), and theorem provers (Coq, Isabelle, KeYmaera X).

2.3 Example 1: Swap

We now demonstrate the benefits of FM thinking and explain a pathway through the different levels via a simple example: swapping two variables.

Figures 1 to 3 show three programs for swapping a pair of numerical constants. They are semantically equivalent in the sense that they satisfy the same sequential specification. Given a precondition $x = m$ and $y = n$, they guarantee the postcondition $x = n$ and $y = m$ for any numerical constants m and n. The first solution is trivial and, in fact, the first solution that students usually learn. The second solution would be obvious to any student familiar with parallel (or multiple) assignment statements, e.g., as supported by Python. However, the third solution is much less obvious and allows a swap to be performed *without* using a temporary variable or a parallel assignment. Convincing oneself that the program implements swap via testing alone would be problematic — which inputs should we try? Does it matter if x and y are integers or reals? Using some simple assertional or symbolic evaluation reasoning, it is quite straightforward to check that the program is indeed correct.[7]

Level 1. A student can be taught to annotate code simply by writing down assertions in natural language, starting with the assumption that x and y are initialised to some arbitrarily chosen values m and n (see Figure 4). The assertion after each line of code is easy to justify from the previous assumption, via elementary "forwards" reasoning, and does not require any mathematical knowledge. The only abstract concepts that students must understand are that

- a *state* maps variables (or locations) to values and
- an *assignment statement* `x:= e` evaluates `e` in the previous state and changes the value of `x` to the evaluated value to generate a new state.

Level 2. At this level, natural language assertions can be replaced by Boolean operators (see Figure 5). For this simple example, on paper, there is seemingly little gain here. However, in more complex examples, students might lose confidence in the correctness of the natural-language assertions they have written. In these cases, they can convert the commented assertions in terms of Boolean operators to checkable (runtime) assertions. In Java, for instance, the penultimate expression in Figure 4 could be augmented with

```
assert x == m + n && y == m;
```

Students now have a tool to not only reason about code but to check their reasoning in parallel with their testing. They will soon discover the limits of that approach, though. First, because testing is incomplete, it might miss a particular case in which an assertion fails. Second, more complex code will require other constructs from predicate logic, such as implication and quantifiers.

[7]Here, we assume a simple model in which overflow is not taken into account. This is still a valuable stepping stone towards teaching more detailed models in which $x + y$ could result in an overflow error.

```
// Here: x = m and y = n for some
// arbitrarily chosen numerical constants
// m and n
x := x + y;
// Here: x = m + n and y = n
// since x is x plus the value of y
//       y is unchanged
y := x - y;
// Here: x = m + n and y = m + n - n = m
// since y is x minus the value of y
//       x is unchanged
x := x - y;
// Here: x = m + n - m = n and y = m
// since x is x minus the value of y
//       y is unchanged
```

Fig. 4. Natural language assertions for the program in Figure 3.

```
// {x = m ∧ y = n} for
// some arbitrarily chosen numerical
// constants m and n
x := x + y;
// {x = m + n ∧ y = n}
y := x - y;
// {x = m + n ∧ y = m}
x := x - y;
// {x = n ∧ y = m}
```

Fig. 5. More concise formulation of the natural-language assertions for the program in Figure 3.

$\{x = m \land y = n\}$
```
x := x + y;
```
$\{x - y = m \land y = n\}$
```
y := x - y;
```
$\{y = m \land x - y = n\}$
```
x := x - y;
```
$\{y = m \land x = n\}$

Fig. 6. Proof outline for the program in Figure 3.

```
method swap(m : real, n: real){
    var x := m;
    var y := n;
    x := x + y;
    y := x - y;
    x := x - y;
    assert(x == n && y == m);
}
```

Fig. 7. Dafny proof of the program in Figure 3.

This might pique their interest in techniques and tools that overcome both limitations: those of Level 3.

Level 3. After learning about Hoare logic or weakest preconditions, students might end up with a different derivation (Figure 6) because they know that reasoning is often simpler when one starts from the end of the execution and works backwards towards the beginning. They also understand that for such a simple program, they do not need to work so hard. They can simply open up a verifier and get the calculator to do the work. Using tools such as Dafny (Figure 7) one only needs to provide the pre- and the post-condition, and the prover infers all the relevant intermediate assertions. Of course, we only use Dafny as an example, but tools such as Why3 [Blazy 2019], Logika [Robby and Hatcliff 2021], and Whiley [Pearce et al. 2018] are also commonly used tools for deductive verification at Level 3. For teaching on Level 3, the choice of language is constrained by the availability of such tools. The complexity and size of even small typical examples exceed what can be mastered without a tool. The tools permit focus on the thinking and enable the students to tackle more challenging problems.

2.4 Example 2: Iterative Binary Search

We now consider one of the fundamental recurring problems in CS: finding an element in a collection. Binary search is an efficient solution to this problem (assuming an already sorted input) and is taught to all CS students. Yet it is a subtle algorithm and challenging to get right, even for professional programmers.

In 1986, John Bentley used "Binary Search" as an exercise for a group of more than 100 professional programmers he was teaching. This is what he reported:

> *I've assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: ninety percent of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found). I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult: in the history in Section 6.2.1 of his Sorting and Searching, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.*

What has changed in those almost 40 years since 1986? For the "average programmer", almost nothing: the "almost 90% found bugs" is still with us today in 2023. (Ask anyone who has taught second-year undergraduates.) FM, which started to develop in the 1970s, was supposed to solve that problem. It did not. One reason is that it was regarded as computing "theory", of little relevance to practical programmers and requiring mathematical knowledge which –it is true– is not necessary to write everyday programs. Formal *Logic* is not required to write a correct Binary Search.

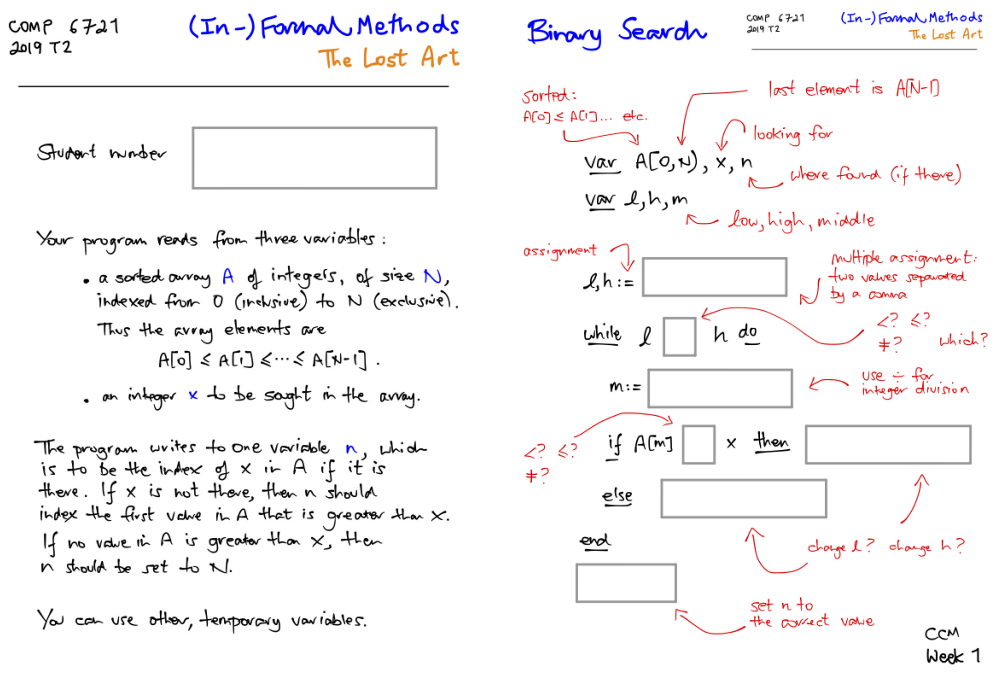
Figure 8 describes an exercise provided to second-year (and later) students at the University of New South Wales, Australia. Students are provided with a description of the algorithm (Figure 8, left) as well as a template for solving it using a `while` loop (Figure 8, right). The exercise is for students to complete the algorithm themselves and then convince themselves that they got it right. Unfortunately, only 10% of the students across multiple cohorts managed to fill in all of the blanks correctly.

To understand the algorithm thoroughly, given that the main work is performed by a `while` loop, students must additionally be taught to think about loop invariants to describe properties that hold before executing each loop iteration and the properties that hold when the loop terminates. The invariant needed for binary search is actually straightforward once a student understands that ℓ and h are used to track the indices of A that have been considered by the algorithm. The value x, if it exists in A, must be between ℓ and h. Thus, the invariant must ensure that:

— the prefix of A up to (but not including) ℓ comprises values that are strictly less than x
— the suffix of A from h onwards comprises values that are at least x (possibly equal)

At Level 1, a student can be taught to include reasoning as in Figure 9, which describes when the program terminates and the properties that are true at different points in the program's execution. It includes subtleties such as the fact that setting m via integer division means that m may be unchanged when $\ell = h - 1$. The assertion for the true branch of the test $A[m] < x$ is sufficient to argue that it is safe to discard the elements $A[0], \ldots, A[m]$ from consideration and the false branch argues the same for $A[m+1], \ldots, A[N-1]$. It is also reasonably straightforward (perhaps using some sketches on a piece of paper) that updating ℓ to $m+1$ in the true case and h to m in the false re-establishes the invariant.

Level 2 skills are a stepping stone to Level 3. For this exercise, students can use their understanding of predicate logic to tighten up the invariants and assertions. For instance, what do the


COMP 6721
2019 T2

(In-) Formal Methods
The Lost Art

Student number

Your program reads from three variables:

- a sorted array A of integers, of size N, indexed from 0 (inclusive) to N (exclusive). Thus the array elements are
  A[0] ≤ A[1] ≤ ··· ≤ A[N-1] .
- an integer x to be sought in the array.

The program writes to one variable n, which is to be the index of x in A if it is there. If x is not there, then n should index the first value in A that is greater than x. If no value in A is greater than x, then n should be set to N.

You can use other, temporary variables.

Binary Search

```
var A[0,N), x, n
var l,h,m
l,h := ☐
while l ☐ h do
    m := ☐
    if A[m] ☐ x then ☐
    else ☐
end
☐
```

In the 2020’s, still only about 10% of students fill this in correctly:
not much has changed since Bentley’s 1986 article. *Why not?*

In the 2020s, still only about 10% of students fill this in correctly:
not much has changed since Bentley’s 1986 article. *Why not?*

Fig. 8. Aiding Levels 1 and 2: Binary search teaching template.

terms “upto”, “from” and “at least” mean exactly? There are choices here: for students familiar (and comfortable) with predicate logic, one could simply write assertions in the style of the invariants from Figure 11. An alternative may be the approach in Figure 10, where the informal assertions from Level 1 are phrased more rigorously.

At Level 3, students can be expected to progress this further by encoding their correctness arguments in Dafny (see Figure 11). The Dafny invariants are from Figure 9, whereas the postcondition (encoded using the `ensures` keyword) captures the requirements described in Figure 8 (left). The proof of Figure 11 in Dafny is *completely automatic*. Once a solution has been found, a curious student at this level is likely to spend *less* time on verification and *more* time on experimentation, since it is straightforward to try out other variations of the algorithm and ask Dafny to check that these are correct. Some possibilities that a student may wish to try are already suggested in Figure 8 (right). Through such experimentation, a student would see that there are two solutions to the final assignment, i.e., that `n := h` and `n := l` both produce the same result. The student can introduce and prove that “`assert(l == h)`” holds immediately before the final assignment to understand why the two variations are both correct.

3 Pathways to Rigorous Reasoning

Almost all CS curricula ensure that students are taught an adequate level of mathematical foundations as well as introductions to programming, algorithms and data structures. These then

```
l, h := 0, N;
// Terminate when l = h
// Continue as long as l < h
while l < h
// Inv: prefix A upto l is less than x
//      suffix A from h is at least x
   m := (l+h)/2;
// Here: l ≤ m < h
   if A[m] < x then
// Here: A[m] < x and m < h
//       so safe to move l to m + 1
      l := m+1
   else
// Here: x ≤ A[m]
//       so safe to move h to m
      h := m
// Here: l = h
//    prefix of A upto l is less than x
//    suffix of A from l is at least x
n := l
```

Fig. 9. Level 1: Reasoning about binary search.

```
l, h := 0, N;
// {l ≤ h}
while l < h
// Inv: (a < x for any a ∈ A[0:l]) ∧
//      (x ≤ a for any a ∈ A[h:N])
   m := (l+h)/2;
// {l ≤ m < h}
   if A[m] < x then
//    {A[m] < x ∧ m < h}
//    so safe to move l to m + 1
      l := m+1
   else
      // {x ≤ A[m]}
      // so safe to move h to m
      h := m
// {l = h ∧
//  (a < x for any a ∈ A[0:l]) ∧
//  (x ≤ a for any a ∈ A[l:N])}
n := l
```

Fig. 10. Level 2: Intermediate-level assertions without using predicate logic, where $A[i : j] = \{A[k] \mid i \leq k < j\}$.

```
method BinarySearch(A: array<int>, x: int)
returns (n: int)

requires forall i, j :: 0 <= i < j < A.Length ==> A[i] <= A[j]

ensures 0 <= n <= A.Length
ensures n < A.Length ==> (A[n] == x) || (A[n] > x && (n > 0 ==> A[n-1] < x))
{ var l, h := 0, A.Length;
    while l < h
    invariant 0 <= l <= h <= A.Length
    invariant forall i :: 0 <= i < l ==> A[i] < x
    invariant forall i :: h <= i < A.Length ==> x <= A[i]
    {
        var m := (l + h) / 2;
        if A[m] < x {
            l := m+1;
        } else {
            h := m; }
    }
    n := l;
}
```

Fig. 11. Level 3: Binary search in Dafny.

specialise in later years to particular applications and CS knowledge areas, e.g., databases, security, concurrency, networks and **artificial intelligence (AI).** Depending on what is taught in later years, the foundational mathematics taught covers topics such as discrete mathematics, logic, probability theory, linear algebra and calculus. However, integrating mathematical foundations with CS topics can often be challenging: CS students often find that *general* mathematical foundations do not motivate them. Simultaneously, within CS courses, it may not always be obvious which mathematical concepts apply to a particular knowledge area.

What level of mathematics is right "for all programmers" is not our topic here: for example, FM thinking should also address the dynamics or behaviour of systems, which is not prominent in discrete mathematics. Instead, we focus on showing that FM thinking builds a bridge, a pathway between mathematics and CS. Once students have some command of the underlying mathematics, these very concepts can be used to argue rigorously about the correctness of the CS systems being built. FM thinking may be systematically incorporated in the curriculum *without* increasing the hours spent teaching. We exemplify one such pathway for software development in Section 3.1. We discuss the role of FM thinking in other areas of CS in Section 3.2.

3.1 Case Study: Software Development

FM thinking prepares a student to think of software as an implementation of a specification instead of only as an executable. This helps answer questions that students often have about program development, e.g., "What is the contribution of Automata Theory or Complexity Theory, which are currently cornerstones of the standard curriculum?" More particularly, "What courses facilitate the transfer of the knowledge and skills of these courses to the software development routine?" In a more technical setting, "What courses help to bridge the gap between the mathematical language of mechanical and electrical engineers and the discourse of software engineers?". Below, we outline how the different levels of FM thinking can be integrated into a degree curriculum that teaches software development.

3.1.1 Level 1. The static "What's True Here" approach can be injected into a first-year programming course already in a rather lightweight fashion while leaving all else in that course as it is. Lightweight FM thinking can also be readily incorporated into a second-year programming course in which the focus is on larger, structured programs and where object-oriented concepts are often taught. The key to writing such programs well is modularity and, as Liskov points out in the preface to her book *Program Development in Java* with John Guttag [Liskov and Guttag 2001], modularity requires abstraction. The book, based on a second-year programming course taught at MIT, introduces programming in Java through the lens of abstraction.

Methods are introduced in the context of procedural abstraction using natural language pre- and postconditions. That style of specification, referred to as "Design by Contract" [Meyer 1988], naturally opens a discussion on when exceptions should be used for incorrect inputs in a defensive style of programming. It also allows students to easily check when subclasses are also subtypes using Liskov's substitution principle [Liskov and Wing 1994]. Lack of understanding of this principle is a well-known problem in real-world software development [Van Roy and Haridi 2004, p. 521].

Classes are introduced in the context of data abstraction, i.e., information hiding, using natural language abstraction functions and class invariants. Once students understand these concepts, it is easy to explain to them why they should not allow access to private variables and should not expose these variables via aliasing with method parameters and return values. This helps students avoid these somewhat subtle errors in their implementations.

3.1.2 Level 2. Isolating propositional and predicate logic, discrete mathematics and/or graph theory as separate courses in the second year (or first year) might not work well. To the student, the motivation for "doing this stuff" has to be clear. When integrated with courses on data structures or algorithms, or combined with modelling and programming challenges, students will reach the point where they have an intrinsic need to bring their FM thinking to the next level. Logic and discrete mathematics will bring the concepts and the techniques to express these needs and assess their initial and final answers.

An example of using Level 2 FM Thinking in an object-oriented programming course is given in the popular textbook *An Introduction to Programming and Object-Oriented Design Using Java* by

Niño and Hosch [2008]. Like Liskov's book, their text, based on a course run at the University of New Orleans, introduces students to the "Design by Contract" paradigm of programming. Again, they use this to discuss the differences with defensive programming, particularly with respect to error handling and exceptions.

In that book, pre- and postconditions are written using Java syntax for Boolean conditions (incorporating English, e.g., "implies" or "for all", when greater expressiveness is required). The specification is incorporated into standard Javadoc comments using custom Javadoc tags, @require and @ensure (although one can also use standard Javadoc tags @param for preconditions, and @return for postconditions). The book shows how preconditions written with Java syntax can be implemented with runtime assertions and how this is useful while developing, debugging and testing software systems. Loop invariants are introduced using the same notation, and the book illustrates how they can be used to (informally) verify correctness of a loop-based program.

3.1.3 Level 3. In the third year, selected courses may be devoted to the theory behind FM thinking, i.e., formal methods. They can be optional courses, such as Process Algebra at TU Eindhoven, but also courses taught in combination with other topics, such as Rigorous Software Engineering at ETH mentioned above. Further specialisation can take place in the Master of Science curriculum.

Following on our theme of FM thinking in second-year programming courses introducing object orientation, Rustan Leino's book, *Program Proofs* [Leino 2023], introduces students to a specification *idiom* for classes that formally captures the concepts of data abstraction in Dafny. In addition to information hiding, the idiom provides constraints to avoid aliasing in, and between, objects of a class. These constraints ensure that Dafny programs using objects of such a class can be verified as expected. By formalising precisely what is required in terms of avoiding aliasing, they provide students with a deep understanding of the associated issues and, hence, a solid basis for developing correct object-oriented programs.

3.2 FM Thinking in Other Streams

The above is mostly focussed on programming, and it is there that FM thinking can have the biggest impact: helping students to model the computational question they are addressing and to understand the code they are writing, consequently producing better programmers. Verifiers such as Dafny can certainly be introduced in later programming courses (if not in the first year) to allow students to take their FM thinking into the real-world context. They do not have to verify entire programs, but they can just prove the tricky methods correct. More generally, FM thinking can also be applied in almost all phases of the software life cycle:

(i) for system modelling and requirements elicitation in the analysis phase,
(ii) for the specification of functional requirements in the product design phase,
(iii) for establishing correctness of subprograms, the writing of clean code, and writing clear documentation in the software development phase,
(iv) for property verification to complement testing in the testing phase, and
(v) for application of FM thinking for system correction and adaptation in the maintenance phase.

At the heart of the problem is the need for precision and rigour in modelling and analysing computer systems and software. Examples of CS streams in which FM are commonly used include the following.

Software engineering. FM can be used to specify and verify software requirements, design, and implementation. The **Software Engineering Body of Knowledge (SWEBOK)** introduces FM as complementing software engineering techniques based on heuristics, prototyping and agile methods. In particular, they ensure that "*the software model can be checked for consis-*

tency (in other words, lack of ambiguity), completeness, and correctness in a systematic and automated or semi-automated fashion." [Bourque and Fairley 2004].

Computer security. FM provides mechanisms for analysing and verifying security protocols and systems, ensuring that the security protocols are free from flaws and that security mechanisms are correctly implemented. As stated in the **CyberSecurity Body of Knowledge (CyBOK)**: "*FM teaches how one can develop a precise specification of (a) the system at an appropriate level of abstraction, such as design or code, (b) the adversarial environment that the system operates in, and (c) the properties, including the security properties, that the system should satisfy.*" [Rashid et al. 2021].

Programming languages and compilers. FM is used to specify and verify programming language semantics, type systems, and compiler correctness, helping ensure that programming languages are well defined and that compilers produce correct code. FM is at the heart of modern programming languages and every course must include FM-based foundational underpinnings. Some recent courses go even further. For example, the Software Foundations series [Pierce et al. 2010] uses the Coq proof assistant to rigorously describe both the features of the programming languages being developed and the algorithms that are implemented in these languages.

Computer networks. FM is used to model and verify network protocols and systems, ensuring that network protocols are correct, efficient, and secure [Bjørner et al. 2015]. This includes checking the consistency and safety of configurations on virtual and physical resources. Is the network free of loops and/or black holes? Is the traffic in logically different networks correctly isolated? Do new or updated configurations conform to properties of the network and not break the consistency of existing networks? Such reasoning is increasingly needed with both the control and data planes being moved under software control.

4 Reflections on the ACM 2023 Knowledge Areas

The CS community is undergoing an effort to develop a new CS standard, called CS2023 [ACM 2023]. The CS field is organised into 16 areas. In the following, we discuss the areas we believe are more related to FM (or at least should be).

Algorithmic foundations. The emphasis here is to present and use the principal data structures and classical algorithms, strategies to construct algorithms, complexity and computability theory.[8] In the core, there are suggestions of addressing invariants (in loops, search algorithms, etc.). However, none of the competencies of this area involves the ability to reason about the *correctness* (including the termination) of algorithms. Although there is a suggestion of performing complexity analysis and even reason about environmental and social impacts of the presented algorithms, again, little is said about correctness. Yet students should learn from the beginning to reason (at least informally) about the correctness of their algorithms. Presenting classical algorithms with an argument for correctness would be a way to introduce this kind of reasoning. Some additional competencies that could be introduced to strengthen a student's knowledge of this area, with respect to Bloom's taxonomy, are the following.

(Analyse level) Precisely define the problem an algorithm is supposed to solve, including expected inputs, outputs and requirements, and construct a solution (from scratch, using existing techniques, and/or adapting and transforming existing solutions).

(Evaluate level) Understand and explain the behaviour of an algorithm, and justify why it satisfies the given requirements.

[8] https://csed.acm.org/algorithms-and-complexity/

Parallel and distributed computing. This knowledge area addresses topics such as parallelisation of programs, dependencies, orderings, atomicity, consensus, progress, deadlocks, faults, safety and liveness.[9] There is no explicit mention of FM, but in many places examples of learning outcomes (suggested for core CS) are to "Write a program that correctly terminates when all of a set of concurrent tasks have completed" and to "Specify a set of invariants that must hold at each bulk-parallel step of a computation". Here, FM and testing play clear and complementary roles: the inherent non-determinism of "Parallel and distributed computing" systems makes it difficult to reproduce situations when testing; yet rigorous testing remains important to ensure that no bugs have been introduced when producing an implementation from the verified system.

Notions of correctness and termination, as well as how to reason rigorously about programs, therefore must be studied beforehand. Moreover, rigorous semantics is crucial here, since minimal misinterpretations can give rise to unexpected behaviours. This area states as prerequisites logic, discrete mathematics, and foundations of software engineering. However, none of them in the current status of the ACM standard provides the ability to be able to understand and justify correctness of computational systems. Notions of correctness of parallelism and distribution combined with established techniques of reasoning (and proof) could advance a student's competencies. Some examples in terms of Bloom's taxonomy are as follows.

(Analyse level) Describe adequately the expected behaviour of parallel and distributed programs and systems, in particular, being able to assess the resulting nondeterminism and limited observability of state.

(Evaluate level) Use rigorous understanding of scheduling, interleaving, and communication primitives to argue why a parallel or distributed program exhibits expected behaviour.

Software development fundamentals. This knowledge area brings together fundamental concepts and skills related to software development, focusing on concepts and skills that should be mastered early in a CS curriculum, typically in the first year. This includes fundamental programming concepts and their effective use in writing programs, use of fundamental data structures which may be provided by the programming language, basics of programming practices for writing good-quality programs, and some understanding of the impact of algorithms on the performance of the programs.[10] However, there is no mention of FM in the proposal or even of informal reasoning about correctness.

We have described the application of FM thinking in "software development fundamentals" in detail in Section 3. Algorithms should not be detached from reasoning about their correctness. One should learn from the very beginning how to specify requirements and justify why these are met by the proposed algorithm. The suggested competencies for "algorithmic foundations" also apply here.

Surprisingly, the ACM 2023 proposal also does not explicitly mention *software architecture*. But there is a natural progression in programming skills that requires knowledge of software architecture. At the same time, software architecture provides more motivation for FM thinking: abstraction and specification, formal or informal, are central to software architecture. The Liskov substitution principle is considered a cornerstone of modern software architecture. To appreciate it fully, some understanding of behavioural specification is required. The associated reasoning is strongly related to concerns of correctness and refinement. Finding robust component abstractions is hard without such mental tools.

[9] https://csed.acm.org/parallel-and-distributed-computing/
[10] https://csed.acm.org/software-development-fundamentals/

In general, as common programming languages such as C++ evolve and new ones such as Rust emerge, FM concepts enter the programming mainstream. For instance, currently an effort is under way to incorporate contracts into C++; concurrent programming with mutexes or channels is based on concepts that were first explored in FM; and the use of ownership and lifetimes in Rust provides a proof of race-freedom based on linear logic. FM thinking supports the practical use of such concepts and the necessary programming methodology and it is difficult to imagine being able to master them fully without the thinking that brought them into existence. We teach children counting and algebra before giving them a pocket calculator for very good reasons. The tools at our disposal require a mindset that puts us in control of them. In terms of Bloom's taxonomy, a student's competencies could be improved as follows.

(Analyse level) Similar to algorithmic foundations, with a focus on concrete concepts of modern programming languages such as those described in the preceding paragraph.

(Evaluate level) Judge and explain whether a component abstraction leaks information that breaks the underlying abstraction.

(Create level) Describe and compose abstract component specifications to construct complex software systems.

Software engineering. This knowledge area focuses on teamwork, tools and environments to build software, software design, software construction (documentation and testing), software verification and validation (basically testing).[11] There is an FM module that is suggested as non-core that covers some FM concepts. This module suggests learning outcomes such as being able to "describe the role of formal specification and analysis", "apply some formal technique to a low complexity problem" and "reason about the benefits and drawbacks of FM". In all other modules, only testing is considered as a validation technique.

Testing and FM thinking are not exclusive. Understanding correctness and reasoning about programs can greatly benefit effective testing. For instance, contracts are helpful to assemble larger software artefacts, describing how different components interact without having to consider the details of their implementations. Having specified contracts formally or informally, knowledge about symbolic execution or predicative interpretation of programs supports the creation of relevant test cases. FM thinking can provide a tool for gathering evidence for the correctness of software by reasoning and testing. This should be considered an essential aspect of software engineering for it to be considered engineering. The software engineer considers the software itself an abstract artefact to be subjected to scrutiny but, unlike other engineering disciplines, is in the fortunate situation that the artefact being constructed can be used as a model of itself. A student's competencies could be strengthened in the following ways.

(Analyse level) Document software with the objective of attaining effective communication among software engineers; design tests that pinpoint faults as precisely as possible based on reasoning about contracts and implementations.

(Evaluate level) Given a set of requirements, a specification, an implementation as well as tests and other verification artefacts, justify that the implementation correctly implements the specification and realises the requirements.

Application of FM in software engineering often requires models of the domain (i.e., environment) in which the software is deployed. For example, Gunter et al. [2000] distinguish the domain, requirements, specifications, programs and programming platforms (or machine) as five separate entities and describe a reference model with proof obligations between these different components. Interestingly, in support of the main thrust of our article, they state [Gunter et al. 2000, pg 42]:

[11] https://csed.acm.org/software-engineering/

The proof obligations are just as sensible for natural-language documentation as they are for formal specifications. Moreover, there is no absolute requirement that the proof obligations be met in the sense of automated theorem proving. On the contrary, the applications we studied have benefited significantly just from the clarity of knowing what the objective of a model's component should be, even without formalization, let alone machine-assisted proof. However, our description is precise enough to support formal analyses.

Thus, there is much to be gained from informal arguments even without full formalisation. What we require are frameworks that enable the transfer of informal arguments to formal proofs. While Gunter et al. [2000] describe a software engineering framework, our article describes a framework for CS education.

5 Teaching the Teachers

In order to integrate FM thinking into the curriculum, we must also be able to train the educators at each of the levels. Arguably, to teach Level 3, the educator must be a specialist practitioner. We have argued that teaching Levels 1 and 2 requires much less specialist knowledge. Despite this, the educator here must still understand (and be able to convey) the benefits of applying FM thinking — what information should a student be capturing when documenting "what's true here"? In this section, we discuss how we can get started with this process, namely, how we can build up the expertise within CS so that *every* educator (including non-specialist FM practitioners) develops a sufficient understanding of FM thinking to be able to teach Levels 1 and 2 effectively.

An ideal situation would be one in which —right at their very first contact with procedural programming, i.e., from Day 1 — students are taught not only the operational view of "what assignment statements do" and "how conditionals and loops work", that is, in terms of "putting a value in that memory box" or "jumping (back) to that bit of program code and carrying on from there" *but also* the static view of "What is made true at that point in the program?" Crucially, it would take no more lecture time to do that: there is plenty of room for both.

The quoted statements above refer to two important aspects of a program that explain how and why it functions correctly. It is the kind of information that distinguishes good program comments from pointless ones. Writing good comments is a skill that is difficult to teach and acquire. The teacher requires a methodology that explains what information is important and how the program text should be annotated with it. For instance: procedures are commented in terms of their contracts; in procedure bodies, blocks are commented with what they achieve (i.e., their specification); loops are commented with their invariants (i.e., what essential property is maintained by the loop in order to attain the specified result). While these concepts are treated entirely informally, they come for free. Only at Level 3 would additional cost be incurred through formal precision and tool usage. The outlined approach can easily be extended by other high-level information describing the idea behind an algorithm or efficiency considerations, but it all starts with being able to say what happens.

That said, we cannot easily at the moment arrange for that to be done, because the lecturers themselves have not been taught that "static" view of programming by *their own* teachers, who were not taught that either. However, *we can change that* and reach a state where there is very little additional effort required.

The long-term is what we should be considering. In the short term, we need to ensure that FM thinking is available to the students who will themselves become lecturers — thus not only breaking the vicious cycle of "there are no teachers to teach the teachers" but also ensuring that university graduates from the workforce who do *not* themselves become teachers will nevertheless become part of a workforce that will deliver better software systems as a direct consequence of asking themselves, as they write their programs, “What’s (supposed to be) true here?”.

Teaching FM thinking at the undergraduate level *does not* require that every teacher be an FM specialist. Not every teacher of Programming or Software Engineering has to be an FM researcher. Teaching at Level 1, FM can be taught *without* formal logic or serious discrete mathematics (see [Cormen et al. 2009; Hallerstede et al. 2019; Morgan 2014]). What is important is the *change in mindset*: not writing “`# Add one to it`” next to an assignment statement “`i = i+1`” because your lecturer said you must document your code, but rather writing what that command has made true because it helps you and your colleagues understand the program.

A course forum at the University of New South Wales in 2023 included an unsolicited post — one student to the rest of the class —*“Why do you think [the “what’s true here”] approach is not more common? Such as establishing invariants, pre- and post-condition comments, etc. It seems that it aids in the programming process and should be used more often.”* Another student replies: *“I think the reason that our approach is not common is that in industry we encounter colleagues from diverse backgrounds, and not everyone has an exposure to [this style of documentation/reasoning]. It could confuse them if they read these comments in our code.”* A precis of that brief exchange might thus be that FM thinking, although it improves programming practice, is not common in industry because *those who can do it might confuse others who were not given the opportunity to learn it*!

Thus, a certain amount of “catch-up” training in elementary, informal FM is necessary in order to reach a self-sustaining level. But not much. The teacher must learn to express things informally enough to keep the course sufficiently light for a broad target audience. What is needed for the teacher as well is being able to distinguish between what is relevant to the reasoning and, complementarily, what is not. For the latter, as with all courses, the teacher must have more experience in the material that they are teaching. Thus, for FM thinking, the teachers themselves must have significantly more experience in FM than the level at which they are teaching. Thus, they must themselves have been exposed to more advanced courses in FM to acquire the concepts and tools to separate the wheat from the chaff. Tuition for FM at Level 3, possibly optional in the curriculum, should therefore have its place — if only to teach the teachers.

The goal therefore is “...how FM thinking can (and should) be incorporated into introductory and intermediate courses on programming, software engineering, data structures and discrete mathematics.” The emphasis is (or should be) on FM thinking in a style that can be taught in second- (or even first-) year alongside the usual operational approaches to elementary programming, and it should not take any more teaching time than is used already.

It will, in the short term, however, take more time in teaching preparation: there is no escape from *having to teach the teachers*. Once that is done, it is done forever: they will teach their successors because they will be convinced of the value of the FM-thinking approach. That is why FM thinking has to be in the curriculum, and it should be early. Otherwise, we will never get started.

6 Related Work

Teaching FM has received increased attention in recent years, for example, with the (re-)emergence of specialist workshops dedicated to this topic [Dongol et al. 2019; Dubois and San Pietro 2023; Ferreira et al. 2021]. This includes freely available courses and associated textbooks [Hehner 1993; Pierce et al. 2010]. Here, community interest has been motivated by the development of tools that support formal proofs at an industrial scale [Calcagno and Distefano 2011; Leino 2017], coupled with a growing desire for deeper integration of formal approaches to system development from within industry [Gleirscher et al. 2020]. However, much of this focus has been on tools and teaching methods that support Level 3. Users are expected to learn how to use verification tools (and their associated logics) to be able to prove desired properties of the program.

In a recent survey of 130 experts in formal methods [Garavel et al. 2020], 50% responded with "Not enough attention" when asked "What is your opinion on the level of importance currently attributed to teaching of formal methods at universities?". A further recurring theme was that "education in formal methods is often isolated" with five comments specifically responding that "applications of formal methods should occur in other courses, like databases, algorithms, concurrency, distributed systems, operating systems, security, compilers, and programming languages".

There have also been attempts to introduce FM early in a CS degree curriculum. However, implementing such teaching methods requires significant investment and specialist educators. For example, a course developed at Reykjavik University [Aceto and Ingólfsdóttir 2021] teaches FM to first-year students via an intensive 3-week course but requires the following: "*During those three weeks, students are expected to engage in activities related to the single course they are following every working day for about eight hours per day.*" Such methods are not in line with the goals of CS2023 [Kumar et al. 2023], who have aimed to keep the allocated Core and Knowledge Area hours as low as possible.

Our proposal is much more lightweight and follows on from earlier ideas presented by Morgan [2014]. Unlike Morgan, who focuses on teaching FM-based programming and proofs, this article focusses on FM and its core principles as tools for supporting a broad range of CS topics. Such ideas are already being implemented. For example, Hallerstede et al. [2019] present a recently developed curriculum in which programming is taught using a combination of informal and formal methods. Here, informal reasoning (Level 1) is used to argue correctness of programs. This is later followed by formally encoding the desired properties (Level 2) and checking correctness using an associated tool (Level 3).

Broy et al. [2024] and ter Beek et al. [2024] also present position papers for FM teaching in reference to CS2023. Broy et al. [2024] argue for FM to be deeply integrated into every CS curriculum, describing the importance of such an integration. Additionally, they discuss why FM should be developed into its own knowledge area, instead of being scattered across the CS2023 curriculum guidelines. ter Beek et al. [2024] discuss the importance of formal methods in industry, drawing on insights from surveys and testimonials from industry figures. They highlight the skills learnt from education in FM that are applicable in many different industrial settings.

7 Conclusions

Despite the intrinsic nature of FM within the discipline of CS, there is often a disconnect between what is optimally desired and what is actually taught within most CS curricula. For example, at the University of Toronto, FM is not required in any of its many programs. Instead, it is an optional course that students may take to round out their program. Each year, about 100 students (out of a cohort of 430) choose it — but probably because it has an excellent reputation both by word-of-mouth and course evaluations and not because FM is considered a valuable subject by the curriculum committee. Some students like the course very much; others do not. But all of them are surprised to learn that programming can be as rigorous as mathematical proving and, more than that, it can be formal and machine checkable.

As discussed in Section 5, teaching and encouraging the use of lightweight FM thinking in CS curricula enables students to become better at documenting their code. This has a knock-on effect of making software easier to maintain since the expectations of a developer are present alongside the code, helping address an important challenge in industry: correctness of software evolution.

The lack of integration of FM thinking within a degree program is not a criticism of typical CS curricula or of the ACM curriculum recommendations, since both are merely a reflection of the true state of the discipline. In particular, both reflect the fact that formal tools and techniques are typically not well integrated into the software-development life cycle outside of major technology companies [Chudnov et al. 2018], or those who develop systems for the safety-critical systems domain. When applied, due to scalability issues, FM techniques are used by developers to understand and verify small (critical) fragments of a much larger system. However, as we have seen, adding FM thinking into the curriculum is not hard to do, and it trains students to focus on correctness.

Ultimately, this article aims to complement the work being carried out by ACM CS2023 [ACM 2023; Kumar et al. 2023] and provide a pathway towards integrating FM-based rigour without increasing the hours dedicated to teaching FM (referred to as Core and Knowledge Area hours [Kumar et al. 2023]). This is the ideal time and place to make these adjustments to the way we teach Computer Science. There are mature tools available to support those adjustments and there is as well the possibility of designing a coherent teaching path that provides a clear "exit ramp" for students for whom the introductory portion —still a significant improvement over what is currently in place —turns out to be enough. Importantly, this can be done without displacing the other "engineering" aspects of CS that are already widely accepted to be essential.

Most will agree that the integration of full formal verification into the software-development life cycle is still some way away. Thus, there is indeed an argument to be made that not all software systems need FM thinking at its Level 3. However, they do need it at Level 2 and have for decades: the reality is, and will remain, that real-world software can be, and always has been, unreliable. This article therefore addresses that fundamental question in CS education and asks how one can teach students to be rigorous *without* having to be fully formal at the same time.

ACKNOWLEDGEMENTS

We thank our reviewers for this article who have helped us strengthen our arguments. The first author has also benefitted from discussions with the ACM CS2023 Foundations of Programming Languages Committee.

References

Luca Aceto and Anna Ingólfsdóttir. 2021. Introducing formal methods to first-year students in three intensive weeks. In *Formal Methods Teaching — 4th International Workshop and Tutorial, FMTea 2021, Virtual Event, November 21, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13122)*, João F. Ferreira, Alexandra Mendes, and Claudio Menghi (Eds.). Springer, 1–17. https://doi.org/10.1007/978-3-030-91550-6_1

Association for Computing Machinery ACM. 2023. *CS2023: ACM/IEEE-CS/AAAI Computer Science Curricula.* ACM. https://csed.acm.org/

Lorin Anderson, David Krathwohl, Peter Airasian, Kathleen Cruikshank, Richard Mayer, Paul Pintrich, James Raths, and Merlin Wittrock. 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Pearson.

Ralph-Johan Back. 2009. Invariant based programming: Basic approach and teaching experiences. *Formal Aspects Comput.* 21, 3 (2009), 227–244. https://doi.org/10.1007/S00165-008-0070-Y

Nikolaj Bjørner, Nate Foster, Philip Brighten Godfrey, and Pamela Zave. 2015. Formal foundations for networking (Dagstuhl Seminar 15071). *Dagstuhl Reports* 5, 2 (2015), 44–63. https://doi.org/10.4230/DagRep.5.2.44

Sandrine Blazy. 2019. Teaching deductive verification in Why3 to undergraduate students. In *FMTea (Lecture Notes in Computer Science, Vol. 11758)*, Brijesh Dongol, Luigia Petre, and Graeme Smith (Eds.). Springer, 52–66. https://doi.org/10.1007/978-3-030-32441-4_4

B. S. Bloom, M. D. Engelhart, E. J. Furst, W. H. Hill, and D. R. Krathwohl. 1956. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Vol. Handbook I: Cognitive Domain.* David McKay Company.

Pierre Bourque and Richard E. Fairley (Eds.). 2004. *SWEBOK: Guide to the Software Engineering Body of Knowledge (Version 3.0).* IEEE Comp. Soc.

Géraldine Brieven, Simon Liénardy, Lev Malcev, and Benoit Donnet. 2023. Graphical loop invariant based programming. In *Formal Methods Teaching: 5th International Workshop, FMTea 2023, Lübeck, Germany, March 6, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13962)*, Catherine Dubois and Pierluigi San Pietro (Eds.). Springer, 17–33. https://doi.org/10.1007/978-3-031-27534-0_2

M. Broy, A. Brucker, A. Fantechi, M. Gleirscher, K. Havelund, M. A. Kuppe, A. Mendes, A. Platzer, J. Ringert, and A. Sullivan. 2024. Does every computer scientist need to know formal methods? *Formal Aspects Comput.* (2024). IN THIS ISSUE.

Cristiano Calcagno and Dino Distefano. 2011. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous formal verification of Amazon s2n. In *Computer Aided Verification — 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 430–446. https://doi.org/10.1007/978-3-319-96142-2_26

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

Brijesh Dongol, Luigia Petre, and Graeme Smith (Eds.). 2019. *Formal Methods Teaching - Third International Workshop and Tutorial, FMTea 2019, Held as Part of the Third World Congress on Formal Methods, FM 2019, Porto, Portugal, October 7, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11758)*. Springer. https://doi.org/10.1007/978-3-030-32441-4

Catherine Dubois and Pierluigi San Pietro (Eds.). 2023. *Formal Methods Teaching: 5th International Workshop, FMTea 2023, Lübeck, Germany, March 6, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13962)*. Springer. https://doi.org/10.1007/978-3-031-27534-0

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press.

João F. Ferreira, Alexandra Mendes, and Claudio Menghi (Eds.). 2021. *Formal Methods Teaching —4th International Workshop and Tutorial, FMTea 2021, Virtual Event, November 21, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13122)*. Springer. https://doi.org/10.1007/978-3-030-91550-6

Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. 2020. The 2020 expert survey on formal methods. In *FMICS (Lecture Notes in Computer Science, Vol. 12327)*, Maurice H. ter Beek and Dejan Nickovic (Eds.). Springer, 3–69. https://doi.org/10.1007/978-3-030-58298-2_1

Jeremy Gibbons. 2021. How to design co-programs. *J. Funct. Program.* 31 (2021), e15. https://doi.org/10.1017/S0956796821000113

Mario Gleirscher, Simon Foster, and Jim Woodcock. 2020. New opportunities for integrated formal methods. *ACM Comput. Surv.* 52, 6 (2020), 117:1–117:36. https://doi.org/10.1145/3357231

Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. 2000. A reference model for requirements and specifications. *IEEE Softw.* 17, 3 (2000), 37–43. https://doi.org/10.1109/52.896248

Stefan Hallerstede, Peter Gorm Larsen, Jalil Boudjadar, Carl Schultz, and Lukas Esterle. 2019. On the design of a new software engineering curriculum in computer engineering. In *Frontiers in Software Engineering Education — First International Workshop, FISEE (Lecture Notes in Computer Science, Vol. 12271)*, Jean-Michel Bruel, Alfredo Capozucca, Manuel Mazzara, Bertrand Meyer, Alexandr Naumchev, and Andrey Sadovykh (Eds.). Springer, 178–195. https://doi.org/10.1007/978-3-030-57663-9_12

Eric C. R. Hehner. 1993. *A Practical Theory of Programming*. Springer. https://doi.org/10.1007/978-1-4419-8596-5 Retrieved from http://www.cs.utoronto.ca/~hehner/aPToP/

Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. MIT Press. 609–619 pages.

Daniel Jackson. 2019. Alloy: A language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

Daniel Jackson and Jeannette Wing. 1996. Lightweight formal methods. *IEEE Comput.* 29, 4 (1996), 21–22.

Amruth N. Kumar, Brett A. Becker, Marcelo Pias, Michael Oudshoorn, Pankaj Jalote, Christian Servin, Sherif G. Aly, Richard L. Blumenthal, Susan L. Epstein, and Monica D. Anderson. 2023. A combined knowledge and competency (CKC) model for computer science curricula. *Inroads* 14, 3 (2023), 22–29. https://doi.org/10.1145/3605215

K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning — 16th International Conference, LPAR (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

K. Rustan M. Leino. 2017. Accessible software verification with dafny. *IEEE Softw.* 34, 6 (2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

K. Rustan M. Leino. 2023. *Program Proofs*. MIT Press.

Barbara Liskov and John V. Guttag. 2001. *Program Development in Java - Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley.

Barbara Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

Bertrand Meyer. 1988. *Object-Oriented Software Construction, 1st Edition*. Prentice-Hall.

Carroll Morgan. 2014. (In-)Formal methods: The lost art —A users' manual. In *Engineering Trustworthy Software Systems — 1st International School, SETSS 2014, Chongqing, China, September 8–13, 2014. Tutorial Lectures (Lecture Notes in Computer Science, Vol. 9506)*, Zhiming Liu and Zili Zhang (Eds.). Springer, 1–79. https://doi.org/10.1007/978-3-319-29628-9_1

Jaime Niño and Frederick A. Hosch. 2008. *An Introduction to Programming and Object-oriented Design Using Java*. John Wiley & Sons, Inc.

David J. Pearce, Mark Utting, and Lindsay Groves. 2018. An introduction to software verification with Whiley. In *SETSS (Lecture Notes in Computer Science, Vol. 11430)*, Jonathan P. Bowen, Zhiming Liu, and Zili Zhang (Eds.). Springer, 1–37. https://doi.org/10.1007/978-3-030-17601-3_1

Benjamin Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. (2010). Retrieved from https://softwarefoundations.cis.upenn.edu/

Awais Rashid, Howard Chivers, Emil Lupu, Andrew Martin, and Steve Schneider (Eds.). 2021. *The Cyber Security Body of Knowledge*. University of Bristol. https://www.cybok.org

Robby and John Hatcliff. 2021. Slang: The Sireum programming language. In *ISoLA (Lecture Notes in Computer Science, Vol. 13036)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 253–273. https://doi.org/10.1007/978-3-030-89159-6_17

Peter Van Roy and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. MIT Press. http://www.info.ucl.ac.be/people/PVR/book.html

M. ter Beek, R. Chapman, R. Cleaveland, H. Garavel, R. Gu, I. ter Horst, J. Keiren, T. Lecomte, M. Leuschel, K. Rozier, A. Sampaio, C. Seceleanu, M. Thomas, T. Willemse, and L. Zhang. 2024. Formal methods in industry. *Formal Aspects Comput.* (2024). IN THIS ISSUE.

Received 24 July 2023; revised 27 March 2024; accepted 20 May 2024